REMARKS

In the Official Action mailed on **June 14, 2004**, the Examiner reviewed claims 1-27. The Examiner requested copies of the references cited in the application. Claims 1-8, 10-17, and 19-26 were rejected under 35 U.S.C. §103(a) as being unpatentable over Jennings et al. (USPN 6,442,752 B1, hereinafter "Jennings") in view of Hari et al (USPN 6,574,673 B1, hereinafter "Hari"). Claims 9, 18, and 27 were rejected under 35 U.S.C. §103(a) as being unpatentable over Jennings in view of to Hari as applied to claim 1 above and further in view of Blandy et al (USPN 6,481,006 B1, hereinafter "Blandy").

Request for references

The Examiner requested copies of the references cited in the application. Applicant provides herewith the requested documents.

Rejections under 35 U.S.C. §103(a)

Independent claims 1, 10, and 19 were rejected as being unpatentable over Jennings in view of Hari. Applicant respectfully points out that Jennings teaches **linking an application** within a first environment with a dynamic link library in a second environment (see Jennings, FIG. 6 index 68, FIG. 12, and col. 8, lines 22-28).

In contrast, the present invention is directed to using an **interprocess call** from an application in a first process to a library routine in a second process (see FIG. 2 and paragraph [0040] of the instant application). This is beneficial because it ensures that operations in the first process are isolated from memory and other system resources belonging to the second process so that an error in the first process does not corrupt memory belonging to the second process (see paragraph [0025] of the instant application). There is nothing within Jennings or Hari, either separately or in concert, which suggests using an interprocess call from an application in a first process to a library routine in a second process so that an error in the first process does not corrupt memory belonging to the second process.

Accordingly, Applicant has amended independent claims 1, 10, and 19 to clarify that the present invention uses an interprocess call from an application in a first process to a library routine in a second process so that an error in the first process does not corrupt memory belonging to the second process. These amendments find support in FIG. 2 and paragraphs [0040] and [0025] of the instant application. Dependent claims 7, 16, and 25 have been canceled without prejudice.

Hence, Applicant respectfully submits that independent claims 1, 10, and 19 as presently amended are in condition for allowance. Applicant also submits that claims 2-6 and 8-9, which depend upon claim 1, claims 11-15 and 17-18, which depend upon claim 10, and claims 20-24 and 26-27, which depend upon claim 19, are for the same reasons in condition for allowance and for reasons of the unique combinations recited in such claims.

CONCLUSION

It is submitted that the present application is presently in form for allowance. Such action is respectfully requested.

Respectfully submitted,

By [illegible]

Edward J. Grundler
Registration No. 47,615

Date: June 30, 2004

Edward J. Grundler
PARK, VAUGHAN & FLEMING LLP
508 Second Street, Suite 201
Davis, CA 95616-4692
Tel: (530) 759-1663
FAX: (530) 759-1665

A Case For Embedding The JVM Into Apps

J. P. Morgenthal
June 22, 1998

Microsoft has been extremely successful at getting Windows applications vendors to license the Visual Basic for Applications library and include it in their applications. By embedding this software inside an application, developers can extend functionality using natively exposed objects that only make sense within the application's context. For example, inside of a graphics package a user can add special effects to the current drawing.

Nothing would be a bigger boon to Java than to have vendors follow the lead of Micro-soft and embed a complete Java Virtual Machine (JVM) inside their applications, thus allowing developers to use Java to extend the functionality of their software. Web and database servers provide this functionality, but these servers are nongraphical environments requiring limits on the functionality of the Java source running there.

This requires vendors to license the VM from Sun Microsystems or develop their own clean-room version, a feat unto itself. One company deciding to bite the bullet and follow this path may reap the rewards to become the king of the computer-aided design (CAD) world. Bentley Systems, a maker of CAD, manufacturing and engineering software, has included and extended a JVM inside its MicroStation/J product.

Most design engineers work in a world of their own and share their CAD models through check-in/check-out facilities. The concept of distributed CAD environments and shared CAD models are foreign to many design engineers. By building the JVM directly into MicroStation/J, Bentley is positioning its product to extend its reach easily over the Internet/intranet. Considering the rate of growth of shared multinational engineering projects, this could be a determining factor in deciding which CAD application to purchase.

Consequently, Microsoft's implementation of the JVM could technically include them in the list of companies that have taken th is direction. By offering access to Component Object Model objects from within a Java application, Microsoft has provided developers a way to expose operating system objects via Java. This direction, however, leads to software that is tied to a particular JVM implementation, but as Bentley has done and as Microsoft is in the process of doing, their JVMs can be ported to other platforms, extending the reach of applications built to use these services.

The real benefit of incorporating the JVM is to reuse existing Java software to speed development time of specialized applications.

J.P. Morgenthal is president of NC.Focus, a consultancy focused on distributed computing. He can be reached at jp@ncfocus.com.

TALx86: A Realistic Typed Assembly Language*

Greg Morrisett Karl Crary† Neal Glew Dan Grossman Richard Samuels
Frederick Smith David Walker Stephanie Weirich Steve Zdancewic
Cornell University

Abstract

The goal of typed assembly language (TAL) is to provide a low-level, statically typed target language that is better suited than Java bytecodes for supporting a wide variety of source languages and a number of important optimizations. In previous work, we formalized idealized versions of TAL and proved important safety properties about them. In this paper, we present our progress in defining and implementing a realistic typed assembly language called TALx86. The TALx86 instructions comprise a relatively complete fragment of the Intel IA32 (32-bit 80x86 flat model) assembly language and are thus executable on processors such as the Intel Pentium. The type system for the language incorporates a number of advanced features necessary for safely compiling large programs to good code.

To motivate the design of the type system, we demonstrate how various high-level language features are compiled to TALx86. For this purpose, we present a type-safe C-like language called Popcorn.

1 Introduction

The ability to type-check low-level or object code, such as Java Virtual Machine Language (JVML) bytecodes [10], allows an extensible system to verify the preservation of an important class of safety properties when untrusted code is added to the system. For example, a web browser can check memory safety to ensure that applets do not corrupt arbitrary data. Indeed, the entire JDK 1.2 security model depends crucially on the ability of the JVML type system to prevent untrusted code from by-passing run-time checks that are needed to enforce the high-level security policy.

To support portability and type-checking, the JVML was defined at a relatively high level of abstraction as a stack-based abstract machine. The language was engineered to make type-checking relatively easy. However, the JVML design suffers from a number of drawbacks:

1. Semantic errors have been uncovered in the JVML verifier and its English specification. Much recent work [1, 16, 18] has concentrated on constructing an *ex post facto* formal model of the language so that a type-soundness theorem can be proven. A by-product of this work is that we now know the design could have been considerably improved had a formal model been constructed in conjunction with the design process.

2. It is difficult (or, at the least, inefficient) to compile high-level languages other than Java to JVML. For instance, approaches for compiling languages with parametric polymorphism have generally involved either code replication [2] or run-time type checks [14]. This has even constrained extensions to Java itself [17]. As another example, definitions of languages such as Scheme [8] dictate that tail calls be implemented in a space-efficient manner. However, the limitations of JVML necessitate that control-flow stacks for such languages be explicitly encoded as heap-allocated objects.

3. Although the JVML was designed for ease of interpretation, in practice, just-in-time (JIT) compilers are used to achieve acceptable performance. Since the JIT translation to native code happens *after* verification, an error in the compiler can introduce a security hole. Furthermore, the need for rapid compilation limits the quality of code that a JIT compiler produces.

To address these concerns, we have been studying the design and implementation of type systems for

*This material is based on work supported in part by the AFOSR grant F49620-97-1-0013, ARPA/RADC grant F30602-1-0317, NSF grant CCR-9708915, and National Science Foundation Graduate Fellowships. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not reflect the views of these agencies.

†Carnegie Mellon University

machine languages. The goal of our work is to identify typing abstractions that have general utility for encoding a variety of high-level language constructs and security policies, but that do not interfere with optimization. Such abstractions are necessary even in very expressive contexts such as proof-carrying-code [15].

In previous work [13, 12], we presented a statically typed, RISC-based assembly language called TAL, showed that a simple functional language could be compiled to TAL, and proved that the type system for TAL was sound: well-typed assembly programs could not violate the primitive typing abstractions. In later work, we described various extensions to support stack-allocation of activation records (and other data) [11] and separate type-checking and link-checking of object files [6]. The languages described were extremely simple so as to keep the formalism manageable.

In this paper, we informally describe TALx86, a statically typed variant of the Intel IA32 (32-bit 80x86 flat model) assembly language. The TALx86 type system is considerably more advanced than the type systems we have described previously. In addition to providing support for stack-allocation, separate type-checking and linking, and a number of basic type constructors (*e.g.*, records, tagged unions, arrays, *etc.*), the type system supports higher-order and recursive type constructors, arbitrary data representation, and a rich kind structure that allows polymorphism for different "kinds" of types.

To demonstrate the utility of these features, we also describe a high-level language called Popcorn and a compiler that maps Popcorn to TALx86. Popcorn is a safe C-based language that provides support for first-class polymorphism, abstract types, tagged unions, exceptions, and a simple module system. Ultimately, Popcorn will support other C-like features such as stack-allocated data and "flattened" data structures.

We begin by giving a brief overview of the process of compiling a Popcorn program to TALx86, verifying the output of the compiler, and creating an executable. We then discuss the salient details of Popcorn. Finally, we present the TALx86 type system by showing how Popcorn programs can be translated to type-correct TALx86 code. We close by discussing planned extensions.

The current software release for TALx86 and Popcorn is available at `http://www.cs.cornell.edu/talc`.

2 TALx86 Tools

This section describes how the TALx86 tools (listed in Table 1) are used together to develop safe native programs. As a running example, we assume the Popcorn source for an application is in two files, `foo.pop` and `main.pop`.

First Popcorn compiles each file separately. If there are no syntax or type errors, then six new files are generated: `foo.tal`, `foo_i.tali`, `foo_e.tali`, `main.tal`, `main_i.tali`, and `main_e.tali`. The `.tal` files contain IA32 assembly language with type annotations, as described in Section 4. A `.tal` file also records what values it imports and exports by listing typed *interface* files. Any `extern` declarations are compiled into the corresponding import interface file (`_i.tali`). Non-static types and values are compiled into the corresponding export interface file (`_e.tali`).

Next we can run the TALx86 type-checker (called `talc`) on `foo.tal` and `main.tal` separately. This step verifies that the TALx86 code is type-safe, given the context implied by the corresponding import file. If the Popcorn compiler is implemented correctly, type-checking the individual `.tal` files that it produces will never fail. By running `talc`, however, we are no longer assuming that the Popcorn compiler produces safe code.

The link-verifier checks that multiple `.tal` files make consistent assumptions about the values and types they share. Popcorn code may fail to link-check, just as traditional object files may fail to link, due to missing or multiple definitions. Unlike a traditional linker, the link-verifier also checks that files agree on the *types* of all shared values. Link-verification guarantees that several `.tal` files are type-safe after being linked together. See Glew and Morrisett [6] for the technical details.

The `.tal` files can be assembled and linked with traditional tools. They are compatible with MASM (Microsoft's Macro Assembler) except that MASM fails on long lines. We have developed an assembler without this deficiency.

Finally, to produce a stand-alone executable some additional trusted files are linked. One component is the Boehm-Demers-Weiser conservative garbage collector [3] which is responsible for memory management. There is also a small runtime environment that provides essential features such as I/O. Although the runtime cannot be written in TALx86, the types of its values can, so the runtime is revealed to applications via a typed interface file.

We have described the build cycle for an executable in detail. In practice, the tools compose these steps

TALx86 tools	
talc	Type-checks a TALx86 file.
link-verifier	Verifies that linking a set of TALx86 files together is safe.
assembler	Assembles a TALx86 file to produce a COFF or ELF object file.
popcorn	Compiles Popcorn to TALx86.
scheme	Compiles a small subset of Scheme to TALx86. Written in Popcorn.

Table 1: Components of the TALx86 implementation

by default, providing the programmer a build interface similar to those in traditional, unsafe systems.

Although Popcorn is the only "serious" compiler targeting TALx86 at this time, TALx86 is not specifically designed for Popcorn. In fact, we have written a compiler for a small part of Scheme, thus demonstrating the feasibility of compiling a higher-order, dynamically typed language.

3 Popcorn

In this section, we briefly summarize the features of Popcorn. This discussion provides a starting point for the following section on compiling to TALx86.

The language purposely looks like C [9], but unsafe features, such as pointer arithmetic, the address operator, and pointer casts, are missing. Compiling these features safely would impose a significant performance penalty on all Popcorn code. Popcorn does have several advanced features not in C such as exceptions and parametric polymorphism. It does not have objects for reasons discussed in the Future Work section. In addition, we avoid various Java-style semantic decisions for efficiency reasons. For example, compiling Java correctly requires run-time type checks on array updates, and its precise exception semantics prevents some standard optimizations.

3.1 Control Flow

The basic control constructs of Popcorn, such as `if`, `while`, `for`, `do`, `break`, and `continue`, are identical to those in C except that test expressions must have type `bool`.[1]

Popcorn's `switch` construct differs from C in that execution never "falls through" cases. Furthermore, a default case is required unless the other cases are exhaustive. The argument of a switch test expression can be an `int`, `char`, `union`, or `exception`. For example, we could find the first occurrence of the character `'a'` in an array:

[1]The result type of relational and logical operators is `bool`.

```
int i = 0, answer;
while (true)
  switch arr[i] {
   case 'a': answer = i;
             break; // break from while
   default:  i++;
  }
```

Array subscripts are bounds-checked at run time (see Section 4.4); the above example will exit immediately if `arr` does not contain an `'a'`.

Exceptions may have different types and exception handlers may switch on the name of an exception, as in Java. However, exception names are not hierarchical.

3.2 Data

Currently, the simple types of Popcorn are `bool`, `char`, `short`, `int`, `string`, and `unsigned` variants of the numeric types. We intend to add floating point numbers and long integers soon. Unlike C, strings are not null-terminated. Arrays carry their size to support bounds-checks. A special `size` construct retrieves the size of an array or string.

Popcorn also has tuples which are useful for encoding anonymous structures and multiple return values. The `new` construct creates a new tuple (as well as new `struct` and `union` values). For example, the following code performs component-wise doubling of a pair of ints:

```
*(int,int) x   = new (3, 4);
*(int,int) dbl = new (x.1+x.1, x.2+x.2);
```

Popcorn has two kinds of structure definitions: `struct` and `?struct`. They resemble `struct *` in C. The difference between `struct` and `?struct` is that values of types defined with `struct` cannot be `null`, which is a primitive construct in the language. Values of types defined with `?struct` are checked for null on field access; failure causes the program to exit immediately.

Unions in Popcorn are more like ML datatypes than C unions. Each variant consists of a tag and an associated type (possibly void). For example,

```
union tree
{void Leaf; int Numleaf; *(tree,tree)Node};
```

Any value of a `union` type is in a particular variant, as determined by its tag, and may not be treated otherwise. We use `switch` to determine the variant of an expression and bind the corresponding value to a variable. Continuing our example, we can write:

```
int sum(tree e) {
 switch e {
  case Leaf: return 0;
  case Numleaf(x): return x;
  case Node(x): return sum(x.1)+sum(x.2);
 }
}
```

3.3 Parametric Polymorphism

Popcorn function, `struct`, `?struct`, and `union` declarations may all be parameterized over types. For example, we can define lists as:

```
?struct <'a>list {'a hd; <'a>list tl;}
```

To declare that a variable `x` holds a list of ints, we instantiate the type parameter: `<int>list x`. Explicit type instantiation on expressions is not necessary; for example, `new list(3,null)` has type `<int>list`. Having polymorphic functions means we can write a length function that works on any type of list. Polymorphism is particularly useful with function pointers. For example, we can write a map function:

```
<'b>list map('b f('a), <'a>list l) {
   if (l == null) return null;
   return new list(f(l.hd), map(f, l.tl));
}
```

A call to this function could look like:

```
<int>list x;
...
<string>list y = map(int_to_string, x);
```

4 An Overview of TALx86

In this section, we give an overview of the features found in TALx86 and describe via example how those features may be used. In particular, we show how Popcorn code may be compiled to well-typed TALx86.

TALx86 uses the syntax of MASM for instructions and data, and augments it with syntax for type annotations necessary for verification. The type annotations can be broken into the following classes:

1. Import and export interface information – used for separately type-checking object files.
2. Type constructor declarations – used to declare new types and type abbreviations.
3. Typing preconditions on code labels – used to specify the types that registers must have before control may enter the associated code.
4. Types on data labels – used to specify the type of a static data item.
5. Typing coercions on instruction operands – used to coerce values of one type to another.
6. Macro instructions – used to encapsulate small instruction sequences so that the type-checker treats the sequence as an atomic action.

The most important of these are the typing preconditions on code labels (3). These annotations are of the general form:

$$\forall \alpha_1{:}\kappa_1 \cdots \alpha_m{:}\kappa_m.\{r_1{:}\tau_1, \cdots, r_n{:}\tau_n\}$$

and are used by the type-checker to ensure that, if control is ever transferred to the corresponding label, then registers r_1 through r_n will contain values of types τ_1 through τ_n respectively. The bound type variables, $\alpha_1,\ldots,\alpha_m$, allow the types on the registers to be polymorphic. One must explicitly instantiate a polymorphic precondition before control can be transferred to the corresponding label. As we will see, TALx86 supports different "kinds" of types. Consequently, each type-variable is explicitly labeled with a kind κ so that we may check that only appropriate types are used to instantiate the bound type variables.

Given a typing precondition for a code label, the type-checker verifies that the instructions in the associated code block are type-correct under the assumptions that $\alpha_1, \cdots, \alpha_m$ are *abstract* types, and that r_i has type τ_i. By treating the type variables as abstract types, we ensure that the code will be type-correct for any appropriate instantiation.

In the rest of this section, we assume that the syntax and semantics of MASM instructions and data will be apparent, and focus our attention on the typing annotations and abstractions. We show how various high-level features from Popcorn may be compiled to TALx86. Due to space limitations, we omit discussion of many TALx86 features, including exceptions, static data, higher-order types, and interfaces.

4.1 Basics

Our first example uses a loop to calculate the sum of the first n natural numbers:

```
int i = n+1;
int s = 0;
while(--i > 0)
  s += i;
```

We could translate the above fragment to the following TALx86 code, assuming n is initially in register ecx:

```
    mov     eax,ecx    ; i = n
    inc     eax        ; ++i
    mov     ebx,0      ; s = 0
    jmp     test
body: {eax: B4, ebx: B4}
    add     ebx,eax    ; s += i
test: {eax: B4, ebx: B4}
    dec     eax        ; --i;
    cmp     eax,0      ; i > 0
    jg      body
```

In this example, the label preconditions say the same thing: "control transfer to this code cannot occur unless registers eax and ebx have B4 values (4-byte integers) in them." The type-checker uses these constraints to check that the operands to each instruction in each block are safe.

Assume for our example that we know ecx initially contains a B4. Then after the first instruction, eax also has a B4. The increment is therefore legal; it is not legal to increment pointers. The third instruction puts a B4 in ebx. Hence the verifier is assured that the precondition for jumping to the test label is satisfied. The test label requires a B4 in ebx even though it does not use the value because it transfers control to body which does use it.

Now consider writing a function:

```
int sum(int n) {
  // previous example is the body
  return s;
}
```

Of course, the function must have some way to return to the caller. Assume for the moment that the caller places the return address in register ebp. In the code below, the typing precondition assumes that ecx contains a 4-byte integer and ebp contains a code label with its own precondition. In particular, the type annotation ebp: {eax: B4} should be read, "ebp contains a pointer to code that expects a B4 in eax."

```
sum:  {ecx: B4, ebp: {eax: B4}}
    <as above>
body: {eax: B4, ebx: B4, ebp: {eax: B4}}
    <as above>
test: {eax: B4, ebx: B4, ebp: {eax: B4}}
    dec     eax        ; --i;
    cmp     eax,0      ; i > 0
    jg      body       ; if so, goto body
    mov     eax,ebx    ; otherwise,
    jmp     ebp        ; return s
```

The final jmp verifies because eax contains a B4. (Notice it would verify even without the preceding mov instruction; type soundness does not guarantee algorithmic correctness.) The type on the sum label describes a non-standard calling convention with the argument in ecx, the return address in ebp, and the result in eax. Such calling conventions are typically used for leaf procedures in an optimizing compiler. One way to "call" sum is to use jmp.

```
    mov     ebp,after
    mov     ecx,10
    jmp     sum
after: {eax: B4}
    <code that uses result>
```

The code explicitly moves the return address (after) into ebp, moves the integer argument into ecx, and then jumps to sum. The jump type-checks because the precondition on sum requires an integer in ecx and a return address in ebp that expects an integer in eax.

4.2 Stacks and Function Calls

To support richer and more realistic calling conventions, TALx86 has a control-flow stack abstraction and stack types. The following examples demonstrate how these types are used. For a theoretical discussion, see Morrisett et al [11].

The standard C calling convention on Win32 requires that the return address be placed on top of the stack,[2] followed by the arguments. Before returning, a function pops the return address. The caller is responsible for popping the arguments.[3]

TALx86 describes the shape of the stack as a list of types, where se represents an empty stack and if σ is a stack type, then $\tau::\sigma$ is the type that describes stacks where the top-most element has type τ and the rest of the stack is described by σ. For example,

[2]Stacks "grow" towards lower addresses; the "top" is the lowest address.

[3]Also, ebp is callee-save; we will incorporate this shortly.

`{eax: B4}::B4::B4::se`

is the type of a stack with three elements: a return address expecting a B4 in eax and then two B4 values. If a register points to a stack (as esp generally does), we write `esp: sptr` σ where σ is a stack type.

If we give our sum function the type `{esp: sptr {eax: B4}::B4::se}`, then we can only call sum when the stack contains exactly the return address and the argument. Clearly we would like calls to sum to type-check regardless of the depth of the stack. To overcome this problem, TALx86 supports *stack polymorphism* to abstract portions of the stack. For example, we could assign sum the type:

```
∀ρ:Ts.
{esp: sptr{eax: B4, esp: sptr B4::ρ}::B4::ρ}
```

which says, "for any stack shape ρ, sum can be called whenever esp contains a pointer to a stack with a suitable return address, followed by an integer, followed by a stack with shape ρ." The code associated with sum is verified treating ρ as an abstract type.

Notice that if sum returns by jumping to the given return address, the stack must have the same shape as on input except without the return address. Indeed, a much stronger property holds since sum is type-checked holding ρ abstract: The input stack corresponding to ρ will remain unmodified throughout the lifetime of the procedure [4]. Hence, a caller is assured that sum will not read or modify the caller's local data (or that of its caller, *etc.*).

Returning to our example, `mov eax,ecx` at the beginning of sum would now become, `mov eax,[esp+4]` so as to load the integer argument from the stack into eax. The final jmp would be replaced with retn, which pops the return address and then jumps to it. A call to sum must now have an additional annotation that instantiates ρ with the actual stack type (not including the input argument, which is not part of ρ). A simple example is:

```
main: {esp: se}
    push    42 ; hidden on stack
    push    10 ; input argument
    call    tapp(sum, <B4::se>)
after:
    <code after>
```

The call instruction pushes the return address (after) before jumping, and the tapp instantiates ρ with `B4::se`.

Usually a call will occur in a context where part of the stack is already abstract, so the instantiation of ρ will use a stack variable in scope at the call site. Indeed, ρ can be instantiated with a stack type containing ρ! In this respect, TALx86 supports a form of polymorphic recursion. For example, Figure 1 shows a recursive implementation of sum. The recursive call says that the stack now has one more B4 and return address on it.

We can also use polymorphism to encode callee-save registers into the calling convention. To force sum to preserve the value in ebp, we require that ebp has a value of distinct abstract type α on entry and exit. We would write:

```
∀α:T4 ρ:Ts.
  {ebp: α, esp: sptr{ebp: α, ...}, ...}
```

where T4 means that α can be any 4-byte type. A call would now have to instantiate α and ρ appropriately.

TALx86 supports addition of constants to stack pointers, and values may be written into arbitrary non-abstract stack slots. Thus, it is not necessary to replace a value on the stack via a sequence of pushes and pops; the element can be directly overwritten.

Additional constructs in the stack-typing discipline of TALx86 support other compiler tasks. For instance, to compile Popcorn exceptions, the code generator needs to pop off a dynamic amount of data from the control stack. To support this, TALx86 provides a limited form of pointers into the middle of the stack. These limited pointers are also sufficient to support displays (static links) for compiling languages such as Pascal. However, they are not sufficient to support general stack-allocation of data.

4.3 Memory Allocation

To support general heap allocation of data, TALx86 provides additional constructs that we now explore, beginning with tuples. Recall our Popcorn tuple code from Section 3:

```
*(int,int) x   = new (3, 4);
*(int,int) dbl = new (x.1+x.1, x.2+x.2);
```

At the assembly level, creating a new pair involves two separate tasks: allocating memory and initializing the fields. This TALx86 code corresponds to the preceding Popcorn:

```
malloc  8,<[:B4,:B4]> ; get space for x
mov     [eax+0],3     ; initialize x.1
mov     [eax+4],4     ; initialize x.2
push    eax           ; save x
malloc  8,<[:B4,:B4]> ; get space for dbl
mov     ebx,[esp+0]   ; x   in ebx
mov     ecx,[ebx+0]   ; x.1 in ecx
add     ecx, ecx      ; x.1+x.1 in ecx
mov     [eax+0], ecx  ; initialize dbl.1
mov     ecx,[ebx+4]   ; x.2 in ecx
```

```
int sum(int n) {           sum: ∀ρ:Ts. {esp: sptr{eax: B4, esp: sptr B4::ρ}::B4::ρ}
  if (n==0)                    cmp     [esp+4],0
    return 0;                  jne     tapp(iffalse, <ρ>)
  else                         mov     eax,0
    return n+sum(n-1);         retn
}                          iffalse: ∀ρ:Ts. {esp: sptr{eax: B4, esp: sptr B4::ρ}::B4::ρ}
                               mov     ebx,[esp+4]
                               dec     ebx
                               push    ebx
                                 ; recursive call instantiates ρ using current stack shape
                               call    tapp(sum, <{eax: B4, esp: sptr B4::ρ}::B4::ρ>)
                               add     esp,4
                               add     eax,[esp+4]
                               retn
```

Figure 1: Recursive Function with C Calling Convention

```
add     ecx,ecx         ; x.2+x.2 in ecx
mov     [eax+4], ecx    ; initialize dbl.2
```

The malloc "instruction" is actually a macro that expands to code that allocates memory of the appropriate size. This routine puts a pointer to the newly-allocated space into eax. The verifier then knows that eax contains a pointer to *uninitialized* fields as specified in the typing annotation <[:B4,:B4]>.

Tracking initialization is important for safety because fields may themselves be pointers, and the type system should prevent dereferencing an uninitialized pointer. To do this, the type of every field has a variance, one of u, r, w, or rw, standing for uninitialized, read-only, write-only, and read-write respectively. The type system does not allow uninitialized fields to be read. However, uninitialized fields may be written with a value of the appropriate type, and then the field is changed to a read-write field. Sub-typing allows a read-write field to be used as read-only or write-only.

Here are the first three lines of our example where the comment describes the type that the verifier assigns to eax after each instruction:

```
malloc  8,<[:B4,:B4]> ; ^*[B4^u, B4^u]
mov     [eax+0],3     ; ^*[B4^rw, B4^u]
mov     [eax+4],4     ; ^*[B4^rw, B4^rw]
```

For example, the second type says, "a pointer to a tuple with two fields, an initialized B4, followed by an uninitialized B4." Of course, these pointer types can appear anywhere B4 can, such as in part of a stack type or label type.

TALx86 places no restrictions on the order in which fields are initialized, nor does it require that all fields be initialized before passing the pointer to another function. It is possible for a field to be "initialized" more than once by creating an alias. For example:

```
malloc  8,<[:B4,:B4]>
mov     ecx, eax        ; ecx aliases eax
mov     [eax+0],3       ; init 1st field
mov     [ecx+0],4       ; init it again
```

In this code, when the contents of eax are moved into ecx, ecx is assigned the same type as eax. The two stores thus initialize the same field twice. This aliasing does not lead to a type unsoundness because the two values have the same type. Since the type system does not track aliasing, some semantically meaningful optimizations cannot be expressed in code that type-checks. For instance, the verifier rejects the following code because it assumes that the field [ecx+0] is uninitialized:

```
malloc  8,<[:B4,:B4]>
mov     ecx, eax        ; ecx aliases eax
mov     [eax+0],3       ; init 1st field
mov     ebp,[ecx+0]     ; type error!
```

Though it would be possible to augment TALx86 to conservatively track aliasing, doing so would further complicate the type system. Thus far, we have favored this simpler approach.

Finally, though TALx86 supports explicit allocation and deallocation of stack-allocated objects, it does not support general purpose pointers to stack-allocated objects. In contrast, general purpose pointers to heap-allocated objects are supported, but explicit deallocation is not. Rather, we link the TALx86 code against a conservative garbage collector so that unreachable objects may be reclaimed. To support explicit deallocation would require an extensive change to the type system [5].

4.4 Arrays

Support for arrays in TALx86 is perhaps the most complicated feature in the language. The critical issue is that array sizes and array indices cannot always be determined statically, yet to preserve type-safety, we must ensure that any index lies between 0 and the physical size of the array. TALx86 provides a very flexible mechanism for tracking the size of an array without requiring that the size be placed in a pre-determined position.

Array subscript and update require special macro instructions (asub and aupd) which take an array pointer, the size of the array, an integer offset, and for aupd, a value to place in the array. The macros expand into code sequences that perform a bounds check, exit immediately when the index is out of bounds, and otherwise perform the appropriate subscript or update operation. Because the array bounds checks are not separated from the subscript or update operations, an optimizer cannot eliminate or reschedule them. Also, no pointers into the middle of arrays are allowed by the current type system, further limiting optimization.

To support arrays, the TALx86 type system includes two new type constructors. The first, S(s), is called a *singleton* type, where s is a compile-time expression corresponding to an integer. The primary purpose of singleton types is to statically track the actual integer value of a register or word in memory. For instance, if eax has type S(3), then the value in eax must be equal to 3 (i.e., it is drawn from the singleton set $\{3\}$). As with other kinds of type expressions, integer type expressions can be polymorphic. Thus, if ecx has type S(α), then we cannot determine statically the (integer) value contained in ecx. However, if ebx also has type S(α), then the type system can conclude that the contents of the two registers are equal. The type system treats singleton integer types as subtypes of B4 so that they may be used whenever a B4 is required.

The second new type constructor is of the form array(s,τ^v) where τ is the type of the array elements, v is their variance, and s is a type expression that represents the size of the array. Notice that s could be a constant, in which case the size of the array is known statically, or it could be a type variable, in which case the size of the array is unknown. Furthermore, as with other type expressions, s is a purely *static* construct used only for verification — it is *not* available as a run-time value. As we shall show, this gives us the flexibility to place the run-time array size anywhere we want instead of in some fixed position. Furthermore, if the size of the array can be determined statically, then the size need not be tracked at run-time.

The crucial issue is to enforce the property that only a run-time integer value equal to the size of the array is passed to asub or aupd for the appropriate bounds check. In particular, if the array has type array(s,τ^v), then the integer passed as the size of the array must have type S(s). For example, the following TALx86 code increments index 2 of a size 5 array of B4 values:

```
lab: {eax: array(5, B4^rw), ebx: S(5)}
    mov  ecx, 2
 ; put eax[ecx] into edx.
 ; array size in ebx, element size is 4.
    asub edx, eax, 4, ecx, ebx
    inc  edx
 ; put edx into eax[ecx].
 ; array size in ebx, element size is 4.
    aupd eax, 4, ecx, edx, ebx
```

This example may only be used on arrays of size 5. To support arrays whose size is unknown statically, we must introduce an integer type variable and quantify over it to achieve "size polymorphism":

```
lab:∀s:Sint.{eax: array(s,B4^rw), ebx: S(s)}
```

(The instructions do not need to change.)

Our compiler represents all Popcorn arrays as a pointer to a data structure containing the (run-time) size followed by the array elements. An *existential* type is used to tie the type of the run-time size with the type of the array as in:

```
∃s:Sint.^*[S(s)^r,array(s,B4^rw)]
```

The type reads as "there exists some integer s such that I am a pointer to a structure containing an integer equal to s, followed by s B4 values." Using an existential to package the run-time size with the array, we can pass the data structure to any function, or place it in any data structure and yet maintain enough information that we can always perform a checked subscript or update on the array. Notice that though this is the default representation used by our compiler, it is not required by TALx86. In particular, the run-time size and the underlying array could be "unboxed" when the Popcorn array does not escape.

```
?struct int_list {
  int hd;
  int_list tl;
}
int len(int_list lst){
  int i = 0;
  while (lst != null){
    ++i;
    lst = lst.tl;
  }
  return i;
}
```

```
type    <int_list:T4 = ^.(0)*[B4^{rw},'int_list^{rw}]>

len: ∀ρ:Ts.
   {esp: sptr{eax: B4, esp: sptr 'int_list::ρ}::'int_list::ρ}
   mov    eax, 0                        ; i=0  in eax
   mov    ebx, [esp+4]                  ; lst in ebx
   jmp    tapp(test, <ρ>)
body: ∀ρ:Ts.{esp: ..., eax: B4, ebx: ^*[B4^{rw},'int_list^{rw}]}
   inc    eax                           ; ++i
   mov    ebx, [ebx+4]                  ; lst = lst.tl
   fallthru <ρ>
test: ∀ρ:Ts.{esp: ..., eax: B4, ebx:'int_list}
   coerce unroll(ebx)    ; int_list -> ^.(0)*[B4^{rw},'intlist^{rw}]
   btagi  ne, ebx, 0, tapp(body,<ρ>) ; check if ebx is null (0)
   retn                                 ; otherwise return
```

Figure 2: List of Integers Implementation

When the size of the array is known at compile-time, an optimizer could avoid storing the size entirely.

Finally, there are two ways to create arrays in TALx86. An n-tuple of values, all of some type τ and variance v, may be coerced to an array of type array(n,τ^v). Second, the trusted runtime provides a function which takes an integer n and a value x of type τ and returns an array of size n with each element initialized to x.

We are currently working to eliminate the asub and aupd macros and to expose the bounds checks so that an optimizer could eliminate them. To do so requires supporting a more expressive symbolic language of static integer expressions within the type system and the ability to prove inequalities between such expressions as in Xi and Pfenning [19, 20].

4.5 Sums and Recursive Types

To demonstrate TALx86 sums and recursive types, we now consider implementing a linked list of integers (see Figure 2). There are two critical points here: First, a list is fundamentally a *sum type*: a value of type list is either null or a pointer to a tuple, and we must ensure that the code works in either case. Second, the list type is recursive.

The Popcorn code has a ?struct definition for lists and a len function which calculates a list's length. The TALx86 code has a corresponding type definition and corresponding code. The TALx86 type definition says a value can be coerced to have type int_list if it is either the singleton value 0 (for null) or a pointer to a pair of an integer and an int_list.

Upon entry to the len label, the integer variable i is initialized to 0 and placed in register eax. The list argument is placed in ebx and the code jumps to the loop test. The test coerces ebx from the type int_list to its representation type, namely the corresponding sum type. The next instruction, btagi, is a macro instruction that tests whether ebx is not equal (ne) to 0, and if so, branches to the body. The macro expands into a simple compare and branch. The type-checker verifies that the register being tested has a sum type, and using the value tested against, refines the type of the register. In particular, at the label body, we are allowed to make the stronger assumption that ebx is in fact a pointer, and not null. This assumption allows the mov ebx,[ebx+4] operation to verify, which has the effect of setting ebx to the tail of the list.

Our current Popcorn compiler generates more naïve code. The list is tested for null once as part of the while test, and then again when the tail of the list is selected. However, it is clear that an optimizing compiler can use dataflow analysis to determine that the second check is redundant. What is not as clear is whether an optimizing compiler can easily maintain the appropriate typing annotations.

4.6 Making Types Smaller

The TALx86 type annotations take far less space than we have suggested so far. For example, the verifier allows the typing preconditions to be dropped for certain labels. In particular, labels that serve only as forward branch targets need no typing precondition. The verifier simply re-verifies the corresponding code block for each branch. The restriction to forward branches ensures termination of the verifier.

The verifier also supports type abbreviations so that the common sub-terms of types may be abstracted. For example, Popcorn gives the same type to every `string`. Rather than repeat this type everywhere, Popcorn defines a `str` abbreviation and uses it in place of the unabbreviated form:

```
type <str = ∃s:Sint.^*[S(s)^r,array(s,B1^rw)]>
```

Another source of repetition is the code types. For example, our code types essentially repeat the type of the stack twice, once for the stack and once for the type of the return address. We can abstract the calling convention with a function abbreviation:

```
type <F = fn ret:T4 s:Ts.
     {esp: sptr {eax: ret, esp: sptr s}::s}>
```

For example, the fully expanded type of the polymorphic map function is the rather unwieldy:

```
map: ∀α:T4 β:T4 ρ:Ts.
{esp: sptr
  {eax:('list β),
   esp:sptr(∀ρ':Ts.
           {esp: sptr{eax:β esp: sptr α::ρ'}
                         ::α::ρ'})
           ::('list α)::ρ}
  ::(∀ρ':Ts.
    {esp: sptr{eax:β esp: sptr α::ρ'}
              ::α::ρ'})
  ::('list α)::ρ}
```

but with the above abbreviation becomes:

```
map: ∀α:T4 β:T4 ρ:Ts.
     F ('list β)
       ((∀ρ':Ts. F β (α::ρ'))::('list α)::ρ)
```

which is smaller, more readable, and in practice faster to verify.

5 Summary and Future Work

We have described the currently available tools for producing TALx86, including a compiler for the C-like language Popcorn. Through examples, we have demonstrated how TALx86 ensures the safety of assembly code, even in the presence of advanced structures and optimizations.

Planned extensions to our system will both add tools and increase the expressiveness of the languages. They include:

1. A binary object file format to replace TALx86's current ASCII format. This format will save both space and parsing time. It will also provide a better setting for evaluating verifier performance.

2. Support for floating point and MMX instructions. We do not expect this to be difficult.

3. Support for run-time code generation, as developed by Trevor Jim and Like Hornoff at the University of Pennsylvania [7]. In addition, an extension to Popcorn called Cyclone makes these features available at a higher level. We are currently working through some minor interoperability issues.

4. A more advanced dependent type system to allow bounds-check elimination when it can be proven that it is safe to do so.

5. Support for object abstractions in TALx86. To support objects in TALx86 requires either having object types or typing constructs to translate object types into. Having object types in TALx86 would restrict TALx86 to OO languages compatible with that object model. While a lot of research has been done on translating object types, these translations either sacrifice theoretical properties or introduce run-time overhead. We are currently investigating a new efficient object encoding that involves sub-typing, F-bounded quantification, and self quantifiers.

6 Acknowledgments

We thank Chris Hawblitzel who wrote the first version of Popcorn. Trevor Jim and Luke Hornoff provided invaluable feedback on the system in addition to adding support for run-time code generation.

References

[1] Martín Abadi and Raymie Stata. A type system for Java bytecode subroutines. In *Twenty-Fifth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 149–160, San Diego, California, USA, January 1998.

[2] Nick Benton, Andrew Kennedy, and George Rusell. *The MLJ User Guide*, 1998.

[3] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 18(9):807–820, 1988.

[4] Karl Crary. A simple proof technique for certain parametricity results. Technical Report CMU-CS-98-185, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA, December 1998.

[5] Karl Crary, David Walker, and Greg Morrisett. Typed memory management in a calculus of capabilities. In *Twenty-Sixth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 262–275, San Antonio, Texas, USA, January 1999.

[6] Neal Glew and Greg Morrisett. Type safe linking and modular assembly language. In *Twenty-Sixth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 250–261, San Antonio, Texas, USA, January 1999.

[7] Luke Hornof and Trevor Jim. Certifying compilation and run-time code generation. In *ACM Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 60–74, San Antonio, Texas, USA, January 1999.

[8] Richard Kelsey, William Clinger, and Jonathan Rees. Revised^5 report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33(9):26–76, September 1998. With H. Abelson, N. I. Adams, IV, D. H. Bartley, G. Brooks, R. K. Dybvig, D. P. Friedman, R. Halstead, C. Hanson, C. T. Haynes, E. Kohlbecker, D. Oxley, K. M. Pitman, G. J. Rozas, G. L. Steele, Jr., G. J. Sussman, and M. Wand.

[9] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice-Hall, Englewood Cliffs, NJ 07632, USA, second edition, 1988.

[10] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.

[11] Greg Morrisett, Karl Crary, Neal Glew, and David Walker. Stack-based typed assembly language. In *Workshop on Types in Compilation*, volume 1473 of *Lecture Notes in Computer Science*, pages 28–52, Kyoto, Japan, March 1998.

[12] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language (extended version). Technical Report TR97-1651, Department of Computer Science, Cornell University, Ithaca, New York, USA, November 1997.

[13] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. In *Twenty-Fifth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 85–97, San Diego, California, USA, January 1998.

[14] Andrew C. Myers, Joseph A. Bank, and Barbara Liskov. Parameterized types for Java. In *Twenty-Fourth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 132–145, Paris, France, 1997.

[15] George Necula. Proof-carrying code. In *Twenty-Fourth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119, Paris, France, 1997.

[16] Robert O'Callahan. A simple, comprehensive type system for Java bytecode subroutines. In *Twenty-Sixth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 70–78, San Antonio, Texas, USA, January 1999.

[17] Martin Odersky and Philip Wadler. Pizza into Java: Translating theory into practice. In *Twenty-Fourth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 146–159, Paris, France, January 1997.

[18] OOPSLA'98 Workshop. *Formal Underpinnings of Java*. Vancouver, Canada, October 1998.

[19] Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 249–257, Montreal, Canada, June 1998.

[20] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *Twenty-Sixth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 214–227, San Antonio, Texas, USA, January 1999.

Safe Kernel Extensions Without Run-Time Checking

George C. Necula Peter Lee

School of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213–3891
{necula,petel}@cs.cmu.edu

Abstract

This paper describes a mechanism by which an operating system kernel can determine with certainty that it is safe to execute a binary supplied by an untrusted source. The kernel first defines a safety policy and makes it public. Then, using this policy, an application can provide binaries in a special form called *proof-carrying code*, or simply PCC. Each PCC binary contains, in addition to the native code, a formal proof that the code obeys the safety policy. The kernel can easily validate the proof without using cryptography and without consulting any external trusted entities. If the validation succeeds, the code is guaranteed to respect the safety policy without relying on run-time checks.

The main practical difficulty of PCC is in generating the safety proofs. In order to gain some preliminary experience with this, we have written several network packet filters in hand-tuned DEC Alpha assembly language, and then generated PCC binaries for them using a special prototype assembler. The PCC binaries can be executed with no run-time overhead, beyond a one-time cost of 1 to 3 milliseconds for validating the enclosed proofs. The net result is that our packet filters are formally guaranteed to be safe and are faster than packet filters created using Berkeley Packet Filters, Software Fault Isolation, or safe languages such as Modula-3.

This research was sponsored in part by the Advanced Research Projects Agency CSTO under the title "The Fox Project: Advanced Languages for Systems Software," ARPA Order No. C533, issued by ESC/ENS under Contract No. F19628-95-C-0050. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Advanced Research Projects Agency or the U.S. Government.

Appeared in Proceedings of the Second Symposium on Operating Systems Design and Implementation (OSDI '96), Seattle, Washington, October 28–31, 1996, pp. 229–243.

1 Introduction

In this paper we address the problem of how an operating-system kernel or a server can determine with absolute certainty that it is safe to execute code supplied by an application or other untrusted source. We propose a mechanism that allows a kernel or server—from now on referred to as the *code consumer*—to define a safety policy and then verify that the policy is respected by native-code binaries supplied to it by an untrusted *code producer*.

In contrast to some previous approaches, we do not rely on the usual authentication or code-editing mechanisms. Instead, we require that the code producer creates its binaries in a special form, which we call *proof-carrying code*, or simply PCC. A PCC binary contains an encoding of a formal proof that the enclosed native code respects the safety policy. The proof is structured in such a way that makes it easy and foolproof for any agent (and in particular, the code consumer) to verify its validity *without* using cryptographic techniques or consulting with external trusted entities; there is also no need for any program analysis, code editing, compilation, or interpretation. Besides being safe, PCC binaries are also extremely fast because the safety check needs to be conducted only once, after which the consumer knows it can safely execute the binary without any further run-time checking.

In a PCC binary, the proof is linked with the native code so that its validity guarantees the code's safety. Furthermore, proof-carrying code is tamper-proof; the consumer can easily detect most attempts by any malicious agent to forge a proof or modify the code. Tampering can go undetected only if the adulterated code is *still* guaranteed to respect the consumer-defined safety policy. Another feature of the PCC method is that the proof checking algorithm is very simple, allowing fast and easy-to-trust implementations.

The safety policy is defined and published by the code consumer and comprises a set of proof-formation rules, along with a set of preconditions. Safety policies can be defined to stipulate standard requirements such as memory safety, as well as more abstract and fine-grained guarantees about the integrity of data-abstraction boundaries. To take a simple example, consider the abstract type of file descriptors. In this case, a client is said to preserve the abstraction boundaries if it does not exploit the fact that file descriptors are represented as integers (by incrementing a file descriptor, for example).

Although we have worked out many of the theoretical underpinnings for PCC (and indeed, most of the theory is based on old and well-known principles from logic, type theory [4, 11], and formal verification [5, 6, 8]), there are many difficult problems that remain to be solved. In particular we do not know at this point the most practical way to generate the proofs. We have thus set out to gain some preliminary experience, both to measure the benefits and to identify the practical problems.

In the experiments reported in this paper, we have in fact achieved fully automatic proof generation. In general, however, this problem is similar to program verification and is not completely automatable. Actually, the problem is somewhat easier than verification because we have the option of inserting extra run-time checks (as is done in Software Fault Isolation), which would have the effect of simplifying the proving process at the cost of reducing performance. By "extra", we mean run-time checks that are not intrinsically a part of the algorithm of the extension code. (For example, SFI will actually edit the code and insert "extra" checks; PCC does not normally do this.) Fortunately, we have not yet had any need or desire to insert extra run-time checks in any of our PCC examples. Still, automation of proof generation remains as one of the most serious obstacles to widespread practical application of PCC.

In our main experiment, we implemented several network packet filters [12, 15] in DEC Alpha assembly language [19] and then used a special prototype assembler to create PCC binaries for them. We were motivated to use an unsafe assembly language in order to place equal emphasis on both performance and safety, as well as to demonstrate the generality of the PCC approach. In addition to the assembler, we implemented a proof validator that accepts a PCC binary, checks its safety proof, and if it is found to be valid, loads the enclosed native code and sets it up for execution.

The results of this and other experiments are encouraging. For our collection of packet filters, we are able to automate completely the generation of the PCC binaries. The one-time cost of loading and checking the validity of the safety proofs is between 1 and 3 milliseconds. Because a safety proof guarantees safety, our hand-tuned packet filters can be executed safely in the kernel address space without adding any run-time checks. Predictably, they are much faster than safe packet filters produced by any other means with which we are familiar.

We believe that our early results show that proof-carrying code is a new point in the design space that is worthy of further attention and study. This paper presents an overview of the approach. We begin with a brief overview of the process of generating and validating the safety proofs. Then, we make this more concrete by showing how a safety policy can be defined and proofs created for a generic assembly language. This is followed by a description of our main experiment involving safe network packet filters. The benchmark results provide some preliminary indication that the PCC methodology has the potential to surpass traditional approaches from a safety point of view while maintaining or improving performance. In particular, we show that PCC leads to faster and safer packet filters than previous approaches to code safety in systems software, including Berkeley Packet Filters [12], Software Fault Isolation [23], and programming in the safe subset of Modula-3 [1, 9, 17]. Finally, we conclude with a discussion of the remaining difficulties and speculate on what might be necessary to make the approach work on a practical scale.

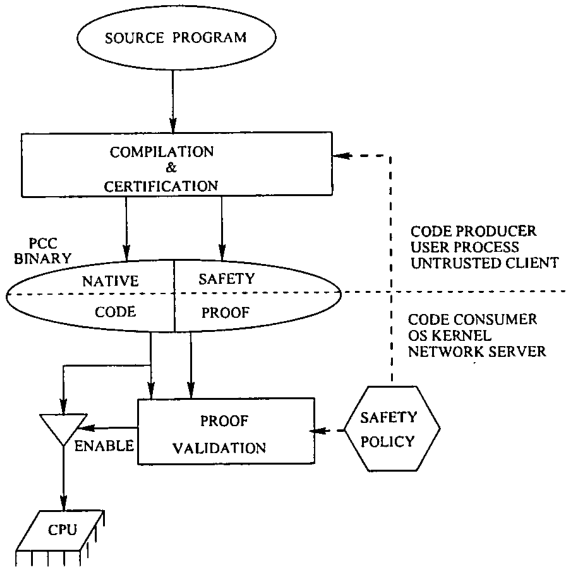

Figure 1: Overview of Proof-Carrying Code.

2 Proof-Carrying Code

Figure 1 depicts the process of generating and using a PCC binary. The process begins with the code consumer defining and publicizing a safety policy. This policy defines formally what is meant by "safety" and also specifies the interface between the consumer and any binary provided by the producer. Taking the policy into account, the code producer compiles (or assembles) and proves the safety of a source program, through a process which we call certification. This results in a PCC binary that can be delivered to the code consumer. Upon receipt, the consumer validates the safety proof enclosed in the PCC binary. Finally, if the proof is found to be valid, the code consumer can safely execute the native-code part of the PCC binary.

The following subsections describe each of these phases in more detail. The whole process is based on concepts from logic, semantics, and type theory, and so the rest of this section is necessarily somewhat technical, with most details beyond the scope of this paper. We will thus attempt to explain only the basic technicalities and key intuitions here. Those readers who would like more details on the underlying theory can find them in a separate technical report [16]. The impatient reader may want to skip ahead to Section 3 where we show, for the case of network packet filters, that proof-carrying code surpasses previous approaches in both safety and performance.

2.1 Defining a Safety Policy

The first order of business is to define precisely what constitutes safe code behavior. We do this by specifying a *safety policy* in three parts:

1. A Floyd-style *verification-condition generator* (also referred to as the VC generator) [6], which is a procedure that computes a predicate in first-order logic based on the code to be certified. We will refer to this predicate as the *safety predicate.*

2. A set of axioms that can be used to validate the safety predicate.

3. The *precondition*, which is essentially a "calling convention" that defines how the code consumer will invoke the PCC binaries.

It is the job of the designer of the code consumer (e.g., the operating system designer) to define the safety policy. In practice, several different safety policies might be used, each one tailored to the needs of specific tasks or services.

We obtain the VC generator by first specifying an *abstract machine* (also called the *operational semantics*), that simulates the execution of safe programs. The abstract machine is not strictly required but it simplifies the design of the safety policy and provides a basis for proving the soundness of the whole approach.

In order to make all of this more concrete, we will now present an example of an abstract machine that specifies a general form of memory safety for the DEC Alpha processor, and then show how the safety policy of a simple resource access service can be defined by a precondition. The VC generator and axioms will then be given in the next subsection.

An abstract machine for memory-safe DEC Alpha machine code

Because the experiments in this paper use the DEC Alpha assembly language, our abstract machine is essentially a high-level formal description of the Alpha architecture [19]. To see how this is done, consider the subset of the Alpha instruction set shown in Figure 2. (Actually, we use a larger subset of the DEC Alpha assembly language in our experiments, but this smaller subset will suffice for presentation purposes.) In this table, n denotes an integer constant and $\mathbf{r}_i$ refers to machine register i. All instructions operate on 64-bit values. For simplicity we allow the use of only 11 temporary and caller-save machine registers (which, for the purpose of this presentation, we rename $\mathbf{r}_0$ through $\mathbf{r}_{10}$). The consequence of this is that programs cannot write into reserved and callee-save registers (according to the standard C calling convention for the DEC Alpha architecture), and are thus trivially safe with respect to these registers.

To define how programs are executed, we define an abstract machine as a state-transition function, the essential core of which is shown in Figure 3. In this specification, the DEC Alpha program is a vector of instructions, Π, and the current instruction is Π_{pc}, where pc is the program counter. The variable ρ denotes the state of the machine registers and memory. The state-transition function maps a machine state (ρ, pc) into a new state (ρ', pc') by executing the current instruction Π_{pc}.

The notation $\rho[\mathbf{r}_i]$ (often abbreviated as $\mathbf{r}_i$) refers to the value of register $\mathbf{r}_i$ in state ρ.[1] The expres-

[1]Valid register values are positive integers in the range 0 to $2^{64} - 1$. This constraint is expressed formally by the equation "$\mathbf{r}_i \bmod 2^{64} = \mathbf{r}_i$", which is applied to all register val-

$$
\begin{array}{lll}
op & ::= & n \mid \mathbf{r}_i \qquad i \in 0 \ldots 10 \\
al & ::= & \text{ADDQ} \mid \text{SUBQ} \mid \text{AND} \mid \text{OR} \mid \text{SLL} \mid \text{SRL} \\
br & ::= & \text{BEQ} \mid \text{BNE} \mid \text{BGE} \mid \text{BLT} \\
instr & ::= & \text{LDQ } \mathbf{r}_d, n(\mathbf{r}_s) \mid \text{STQ } \mathbf{r}_s, n(\mathbf{r}_d) \mid al\ \mathbf{r}_s, op, \mathbf{r}_d \mid br\ \mathbf{r}_s, n \mid \text{RET}
\end{array}
$$

Figure 2: The subset of DEC Alpha assembly language.

sion $\rho[\mathbf{r}_d \leftarrow \mathbf{r}_d \oplus 1]$ denotes the new state obtained from state ρ by incrementing the value of register $\mathbf{r}_d$. So, for example, the Alpha "ADDQ $\mathbf{r}_s, op, \mathbf{r}_d$" instruction is defined by Figure 3 to have the following semantics:

$$(\rho[\mathbf{r}_d \leftarrow \mathbf{r}_s \oplus op], pc + 1)$$

where ρ is the current register and memory state. This specification states that the ADDQ instruction updates register $\mathbf{r}_d$ with the sum of $\mathbf{r}_s$ and op, and also increments the program counter. We use the "circled" operation $\oplus$ to denote two's-complement addition on 64 bits. This operation is defined in terms of the usual integer arithmetic operations as

$$e_1 \oplus e_2 = (e_1 + e_2) \bmod 2^{64}$$

To model the state of memory, we use a pseudo register, called $\mathbf{r_m}$, that gives the content of each memory location. We write $\mathbf{sel}(\mathbf{r_m}, a)$ for the contents of memory address a, and $\mathbf{upd}(\mathbf{r_m}, a, \mathbf{r}_s)$ for the new memory state resulted from writing register $\mathbf{r}_s$ to address a. Memory operations work on 64-bits and the addresses involved must be aligned on an 8-byte boundary.

In the definition of the load and store instructions, there is a crucial difference between the DEC Alpha processor and our abstract machine. The difference is that our abstract machine performs the safety checks that are shown in boxes in Figure 3. For example, consider the definition of the "LDQ $\mathbf{r}_d, n(\mathbf{r}_s)$" instruction:

$$(\rho[\mathbf{r}_d \leftarrow \mathbf{sel}(\mathbf{r_m}, \mathbf{r}_s \oplus n)], pc + 1), \ \textit{if} \ \boxed{\mathbf{rd}(\mathbf{r}_s \oplus n)}$$

The predicate $\mathbf{rd}(a)$ is true when it is safe to read the word at memory address a, which for the DEC Alpha implies that a is aligned on an 8-byte boundary. Similarly, the predicate $\mathbf{wr}(a)$ is true when the address a denotes an aligned location that can be safely read or written. In essence, these checks define what is meant by safety, and more specifically for this example, memory safety. For the purpose of this paper, the predicates $\mathbf{rd}(a)$ and $\mathbf{wr}(a)$ are defined by the safety policy through the precondition, as shown in the next subsection.

ues. Negative values are represented using two's-complement representation.

Mathematically, the abstract machine does not return errors when a $\mathbf{rd}(a)$ or $\mathbf{wr}(a)$ check fails. Instead, the execution blocks because there are no transition rules covering the error cases. In this setting, a program is safe if and only if it runs without blocking on the abstract machine. Of course, the presence of these safety checks means that the abstract machine is not a faithful abstraction of the DEC Alpha processor. However, the purpose of certification is to prove that all safety checks always succeed. If we have a valid safety proof for a program, we know that we can safely execute it on a real DEC Alpha and get the same behavior as on our abstract machine, even though the Alpha does not implement the safety checks.

There are other notable differences between our abstract machine and a real DEC Alpha. For example, to simplify the presentation in this paper, we have restricted all branches to be only forward. Allowing backward branches and loops introduces a number of complications, but is handled in a conceptually straightforward manner through the addition of explicit loop invariants. As it turns out, the packet filter examples we use in our experiments do not have any loops, and so it is not inconvenient to eliminate them here. In a later section we will briefly describe our experiments with looping programs, including a safe IP-header checksum routine.

Another interesting aspect of the abstract machine is the level of abstraction of our specification. We might try to be ambitious and make a complete specification of the DEC Alpha processor. However, this would be extremely complex and probably difficult to trust. And, as a practical matter, for specific tasks such as the ones we are considering, many details and features of the Alpha are irrelevant. This justifies working at a higher level of abstraction above the details of the pipeline, cache, timing, and interrupt behavior.

We can also consider encoding other kinds of safety checks into our abstract machine. For the sake of simplicity, we have specified only a notion of fine-grained memory safety. With some ingenuity, an abstract machine designer can define safety policies involving other kinds of safety, like control over resource usage or preservation of data-abstraction

$$(\rho, pc) \to \begin{cases} (\rho[\mathbf{r}_d \leftarrow \mathbf{r}_s \oplus op], pc+1), & \textit{if}\ \Pi_{pc} = \mathrm{ADDQ}\ \mathbf{r}_s, op, \mathbf{r}_d \\ (\rho[\mathbf{r}_d \leftarrow \mathbf{sel}(\mathbf{r_m}, \mathbf{r}_s \oplus n)], pc+1), & \textit{if}\ \Pi_{pc} = \mathrm{LDQ}\ \mathbf{r}_d, n(\mathbf{r}_s)\ \ \textit{and}\ \ \boxed{\mathbf{rd}(\mathbf{r}_s \oplus n)} \\ (\rho[\mathbf{r_m} \leftarrow \mathbf{upd}(\mathbf{r_m}, \mathbf{r}_d \oplus n, \mathbf{r}_s)], pc+1), & \textit{if}\ \Pi_{pc} = \mathrm{STQ}\ \mathbf{r}_s, n(\mathbf{r}_d)\ \ \textit{and}\ \ \boxed{\mathbf{wr}(\mathbf{r}_d \oplus n)} \\ (\rho, pc+n+1), & \textit{if}\ \Pi_{pc} = \mathrm{BEQ}\ \mathbf{r}_s, n\ \ \textit{and}\ \ \mathbf{r}_s = 0 \\ (\rho, pc+1), & \textit{if}\ \Pi_{pc} = \mathrm{BEQ}\ \mathbf{r}_s, n\ \ \textit{and}\ \ \mathbf{r}_s \neq 0 \end{cases}$$

Figure 3: The Abstract Machine.

$$VC_{pc} = \begin{cases} VC_{pc+1}[\mathbf{r}_d \leftarrow \mathbf{r}_s \oplus op], & \textit{if}\ \Pi_{pc} = \mathrm{ADDQ}\ \mathbf{r}_s, op, \mathbf{r}_d \\ \mathbf{rd}(\mathbf{r}_s \oplus n) \wedge VC_{pc+1}[\mathbf{r}_d \leftarrow \mathbf{sel}(\mathbf{r_m}, \mathbf{r}_s \oplus n)], & \textit{if}\ \Pi_{pc} = \mathrm{LDQ}\ \mathbf{r}_d, n(\mathbf{r}_s) \\ \mathbf{wr}(\mathbf{r}_d \oplus n) \wedge VC_{pc+1}[\mathbf{r_m} \leftarrow \mathbf{upd}(\mathbf{r_m}, \mathbf{r}_d \oplus n, \mathbf{r}_s)], & \textit{if}\ \Pi_{pc} = \mathrm{STQ}\ \mathbf{r}_s, n(\mathbf{r}_d) \\ (\mathbf{r}_s = 0 \Rightarrow VC_{pc+n+1}) \wedge (\mathbf{r}_s \neq 0 \Rightarrow VC_{pc+1}), & \textit{if}\ \Pi_{pc} = \mathrm{BEQ}\ \mathbf{r}_s, n \\ Post, & \textit{if}\ \Pi_{pc} = \mathrm{RET} \end{cases}$$

Figure 4: The Verification-Condition Generator.

boundaries. Once a safety policy is defined, application writers are free to use it to create PCC binaries that guarantee safety.

A sample application and its precondition

The abstract machine as given above describes safety in terms of the abstract notions of readable and writable memory locations. For this to be useful, the code consumer must specify an interface to PCC binaries that identifies the readable and writable memory locations. We do this by specifying a *precondition*, which is a predicate in first-order logic that the code consumer guarantees to be valid when the PCC binary is invoked.

Consider the following simple example. Suppose an operating-system kernel maintains an internal table with data pertaining to various user processes. Each table entry consists of two consecutive memory words—a tag and a data word. The tag describes whether the data word is user writable or not. The kernel also provides a *resource access service* through which user processes are given permission to access their table entry by installing native code in the kernel. To make this possible the kernel invokes the user-installed code with the address of the table entry corresponding to the parent process in machine register $\mathbf{r}_0$. This address is guaranteed by the kernel to be valid and aligned on an 8-byte boundary.

Although this example is somewhat contrived, we can imagine that entries in the table represent capabilities (perhaps file descriptors), and so we would like to provide user-installed code with full access to the correct table entries, while maintaining the integrity of the rest of the table and other parts of the kernel state.

Informally, the safety policy for the resource access service requires that: (1) the user code cannot access other table entries besides the one pointed to by $\mathbf{r}_0$, (2) the tag is read only, (3) the data word is also read only unless the tag value is non zero, and, (4) the code does not modify reserved and callee-saves registers. The last condition ensures that the kernel can safely invoke the user code using a normal C function call.

More formally, the kernel specifies a precondition Pre_r, which states that it is safe to read the tag pointed to by $\mathbf{r}_0$, and that it is also safe to write the data at offset 8 from $\mathbf{r}_0$ if the contents of the tag is not 0. In formal notation, this is written as follows:

$$\begin{aligned} Pre_r = \ & \mathbf{r}_0 \bmod 2^{64} = \mathbf{r}_0\ \wedge\ \mathbf{rd}(\mathbf{r}_0) \wedge \mathbf{rd}(\mathbf{r}_0 \oplus 8) \\ & \wedge\ \mathbf{sel}(\mathbf{r_m}, \mathbf{r}_0) \neq 0 \Rightarrow \mathbf{wr}(\mathbf{r}_0 \oplus 8) \end{aligned}$$

What remains now is to prove for a particular client of the resource access service that all $\mathbf{rd}(a)$ and $\mathbf{wr}(a)$ checks will always succeed, given this precondition and abstract machine. In general, we can also specify a postcondition as part of the safety policy, which would require particular invariants to be valid when the user code terminates. Conceptually, in our example the postcondition is the predicate **true**, meaning that no additional conditions are imposed on the final machine state.

Before moving on to a discussion of the proof generation process, we note that the safety policy we have described here can be thought of as enforcing fine-grained memory protection. In general, one could imagine having much more involved safety requirements. For example, we could change the tag word in the table entry to be a semaphore that the user code must acquire (e.g., atomically test-and-set to zero) before trying to write the data word; furthermore, we could also require (via a simple post-condition) that the code releases the semaphore before returning. Again, for purposes of the current presentation, we stick to the simpler memory-safety requirements.

2.2 Certifying the Safety of Programs

To create safety proofs for a program, we must prove that executing it does not violate any of the safety checks (and the postcondition, if one is given, is also satisfied). Standard techniques exist for building such proofs. Our technique is based on Floyd's verification conditions [6], because they are powerful enough to deal with unstructured assembly-language programs and a broad range of safety invariants. Similar techniques have been used before to verify assembly-language programs [2, 3].

Certification of programs involves two steps:

1. Compute the *safety predicate* for the program. This essentially encodes the semantic meaning of the program in logical form and constitutes a formal statement that the program, when executed, will not violate any safety checks.
2. Generate a *proof* of the safety predicate, written out in a checkable form.

Both these steps are described in the following subsections.

Computing the safety predicate

To compute the safety predicate, we first generate a vector VC of predicates, one for each instruction as specified by the rules in Figure 4. The notation VC_{pc} denotes the predicate for the current instruction. Since the rules specify VC_{pc} in terms of VC_{pc+1}, the verification-condition VC_0 for the beginning of the program can be computed by starting at the end of the program and working back towards the beginning.[2]

[2]This simple approach works because all branches are restricted to be forward-only. We discuss later what happens in the presence of loops.

The rules in Figure 4 are derived in a straightforward manner from the abstract machine specification of Figure 3; in fact, we imagine that experienced kernel and safety policy designers would skip the abstract machine specification and give only the VC generator rules. The notation $P[\mathbf{r}_d \leftarrow \mathbf{r}_s \oplus op]$ stands for the predicate obtained from P by substituting $\mathbf{r}_s \oplus op$ for $\mathbf{r}_d$.

After computing the vector VC, the safety predicate is computed simply by plugging the program Π, precondition Pre, and postcondition $Post$ into the following formula:

$$SP(\Pi, Pre, Post) = \forall \mathbf{r}_0 \ldots \forall \mathbf{r}_{10} \forall \mathbf{r}_{\mathbf{m}} . Pre \Rightarrow VC_0$$

The intuition behind a valid safety predicate is that for any initial state that satisfies the precondition Pre, the code Π starting at the first instruction executes without failure and, if it terminates, the final state satisfies the postcondition $Post$.

```
                               %Address of tag in r0
1    ADDQ   r0,  8, r1         %Address of data in r1
2    LDQ    r0,  8(r0)         %Data in r0
3    LDQ    r2,  -8(r1)        %Tag in r2
4    ADDQ   r0,  1, r0         %Increment Data in r0
5    BEQ    r2,  L1            %Skip if tag == 0
6    STQ    r0,  0(r1)         %Write back data
L1   RET                       %Done
```

Figure 5: DEC Alpha assembly code for resource access. Initially register $\mathbf{r}_0$ holds the address of the tag. The data is at the offset 8 from $\mathbf{r}_0$.

For a concrete example of client code for the resource access service, consider the small program in Figure 5. The overall effect of this program is to increment the data word if it is writable. We first compute VC_0 for this program using the rules in Figure 4; then we compute the safety predicate SP_r using the above formula with the precondition Pre_r and the postcondition **true**. After a few trivial simplifications, the resulting safety predicate is the following:

$$\begin{aligned} SP_r = \forall \mathbf{r}_0 . \forall \mathbf{r}_{\mathbf{m}} . Pre_r \Rightarrow & \ \mathbf{rd}(\mathbf{r}_0 \oplus 8) \wedge \mathbf{rd}(\mathbf{r}_0 \oplus 8 \ominus 8) \\ & \wedge \mathbf{sel}(\mathbf{r}_{\mathbf{m}}, \mathbf{r}_0 \oplus 8 \ominus 8) = 0 \Rightarrow \mathbf{true} \\ & \wedge \mathbf{sel}(\mathbf{r}_{\mathbf{m}}, \mathbf{r}_0 \oplus 8 \ominus 8) \neq 0 \Rightarrow \mathbf{wr}(\mathbf{r}_0 \oplus 8) \end{aligned}$$

Informally, the SP_r predicate says that for all values of register $\mathbf{r}_0$ and states of memory $\mathbf{r}_{\mathbf{m}}$ satisfying the precondition Pre_r, the memory locations $\mathbf{r}_0 \oplus 8$ and $\mathbf{r}_0 \oplus 8 \ominus 8$ must be readable and if the tag (at address $\mathbf{r}_0 \oplus 8 \ominus 8$) is non zero, the data (at address $\mathbf{r}_0 \oplus 8$) must be writable. All these conditions must be true for the code to be safe with respect to the resource access safety policy.

Proving the safety predicate

We have intentionally written the program in Figure 5 in a slightly complicated way, to show that low-level optimizations do not pose significant problems in generating and validating safety proofs. Three of the interesting properties of this program are (1) the instructions are somewhat scheduled, including speculative execution of the load in line 2 and of the addition in line 4, to accommodate the DEC Alpha pipeline latency[3], (2) register $\mathbf{r}_0$ is reused in line 2 to hold the data word instead of the tag address, and (3) even though the precondition is expressed as a function of the value in register $\mathbf{r}_0$, some of the actual memory accesses are done through register $\mathbf{r}_1$. In general, we expect scheduling and register allocation to have no effect on the safety predicate and its proof.

It is a simple exercise for the reader familiar with assembly-language programming to verify that this code is indeed correct with respect to the safety policy. The problem, of course, is how to convince even the most suspicious kernel that this code is absolutely safe. To do this, we must prove the safety predicate according to the rules of first-order predicate calculus extended with two's-complement integer arithmetic. We refer to this set of proof rules as Σ and we write $\vdash_\Sigma SP$ when the safety predicate SP can be proved according to the rules in the set Σ. Most of the rules in Σ are simple. Below we show two of the rules we use, the first being a classical implication-elimination rule from the predicate calculus, and the second a rule about arithmetic:

$$\begin{array}{ll} \vdash_\Sigma Q, & \text{if} \quad \vdash_\Sigma P \Rightarrow Q \quad \text{and} \quad \vdash_\Sigma P \\ \vdash_\Sigma e_1 \oplus e_2 \ominus e_2 = e_1, & \text{if} \quad \vdash_\Sigma e_1 \bmod 2^{64} = e_1 \end{array}$$

The second rule is perhaps a bit surprising because $e_1 + e_2 - e_2 = e_1$ is unconditionally true in integer arithmetic. However, for the machine implementation of arithmetic, this statement is true only if the original value of e_1 is a valid register value.

A large fragment of the proof of the safety predicate for our example program is shown in a proof-tree form in Figure 6. This proof was generated automatically by our PCC system, which incorporates a simple theorem prover. We use vertical dots to stand for extractions of a conjunct from the precondition. You can read the proof tree from top to bottom, interpreting every node as a valid inference of the predicate below the line using the assumptions above the line. For example, in the upper-right corner of the figure the predicate $\mathbf{r}_0 = \mathbf{r}_0 \oplus 8 \ominus 8$ is proved using the arithmetic rule we discussed with the assumption $\mathbf{r}_0 \bmod 2^{64} = \mathbf{r}_0$ extracted from the precondition. Then $\mathbf{wr}(\mathbf{r}_0 \oplus 8)$ is proved using the implication-elimination rule and the hypothesis u of the predicate $\mathbf{sel}(\mathbf{r_m}, \mathbf{r}_0 \oplus 8 \ominus 8) \neq 0$. This hypothesis is introduced at a lower level in the proof tree, at the node labeled u, for the purpose of proving the predicate $\mathbf{sel}(\mathbf{r_m}, \mathbf{r}_0 \oplus 8 \ominus 8) \neq 0 \Rightarrow \mathbf{wr}(\mathbf{r}_0 \oplus 8)$.

[3]These operations are speculative because they are not required if the branch in line 5 is taken.

The guarantee of safety

We use the proof of the safety predicate, written out in an appropriate language (to be described in the next section), as the proof that the code obeys the safety policy. This is justified formally by the *safety theorem*, stated below:

Theorem 2.1 (Safety) *For any program Π, precondition Pre and postcondition Post, if $\vdash_\Sigma SP(\Pi, Pre, Post)$ then for any initial state ρ_0 that satisfies the precondition and for any abstract machine state (ρ, pc) originating from the initial state $(\rho_0, 0)$, one of the following is true:*

1. *The state (ρ, pc) is a final state (i.e. $\Pi_{pc} = \mathrm{RET}$) satisfying the postcondition Post, or*
2. *The execution is not stuck, i.e., there exists a new state (ρ', pc') such that $(\rho, pc) \rightarrow (\rho', pc')$.*

Since the abstract machine gets stuck when there is any violation of an $\mathbf{rd}(a)$ or $\mathbf{wr}(a)$ safety check, this theorem provides an absolute guarantee that a certified program will not have such violations, as long as its execution is started in a state that satisfies the precondition.

The proof of the Safety Theorem is beyond the scope of this paper, but can be found in a separate technical report [16].

2.3 Validating the Safety Proofs

A PCC binary consists of the assembled native code together with an encoding of the proof of its safety predicate. To validate the binary, the code consumer first extracts the native code and then computes its safety predicate using the VC rules. Then, it checks that the safety proof is a valid proof of the safety predicate.

This method ensures safety even if the native code or the proof in the PCC binary is tampered with. If the code is modified, then in all likelihood its safety predicate changes, so the given proof will not correspond to it. If the proof is modified, then either it will be invalid, or else not correspond to the safety predicate. If the code is modified in such

$$
\dfrac{
 \dfrac{
  \dfrac{
   \dfrac{
    \begin{array}{c} Pre_r \\ \vdots \\ \mathrm{rd}(\mathrm{r_0}) \end{array}
    \quad
    \dfrac{\begin{array}{c} Pre_r \\ \vdots \\ \mathrm{r_0} \bmod 2^{64} = \mathrm{r_0} \end{array}}{\mathrm{r_0} = \mathrm{r_0} \oplus 8 \ominus 8}
   }{\mathrm{rd}(\mathrm{r_0} \oplus 8 \ominus 8)}
   \quad
   \dfrac{
    \dfrac{
     \begin{array}{c} Pre_r \\ \vdots \\ \mathrm{sel}(\mathrm{r_m}, \mathrm{r_0}) \neq 0 \Rightarrow \mathrm{wr}(\mathrm{r_0} \oplus 8) \end{array}
     \quad
     \dfrac{
      \dfrac{u}{\mathrm{sel}(\mathrm{r_m}, \mathrm{r_0} \oplus 8 \ominus 8) \neq 0}
      \quad
      \dfrac{\begin{array}{c} Pre_r \\ \vdots \\ \mathrm{r_0} \bmod 2^{64} = \mathrm{r_0} \end{array}}{\mathrm{r_0} = \mathrm{r_0} \oplus 8 \ominus 8}
     }{\mathrm{sel}(\mathrm{r_m}, \mathrm{r_0}) \neq 0}
    }{\mathrm{wr}(\mathrm{r_0} \oplus 8)}
   }{\mathrm{sel}(\mathrm{r_m}, \mathrm{r_0} \oplus 8 \ominus 8) \neq 0 \Rightarrow \mathrm{wr}(\mathrm{r_0} \oplus 8)}\, u
   \quad \cdots
  }{\mathrm{rd}(\mathrm{r_0} \oplus 8 \ominus 8) \wedge (\mathrm{sel}(\mathrm{r_m}, \mathrm{r_0} \oplus 8 \ominus 8) \neq 0 \Rightarrow \mathrm{wr}(\mathrm{r_0} \oplus 8)) \wedge \ldots}\, Pre_r
 }{Pre_r \Rightarrow \mathrm{rd}(\mathrm{r_0} \oplus 8 \ominus 8) \wedge (\mathrm{sel}(\mathrm{r_m}, \mathrm{r_0} \oplus 8 \ominus 8) \neq 0 \Rightarrow \mathrm{wr}(\mathrm{r_0} \oplus 8)) \wedge \ldots}
}{\forall \mathrm{r_0}. \forall \mathrm{r_m}. Pre_r \Rightarrow \mathrm{rd}(\mathrm{r_0} \oplus 8 \ominus 8) \wedge (\mathrm{sel}(\mathrm{r_m}, \mathrm{r_0} \oplus 8 \ominus 8) \neq 0 \Rightarrow \mathrm{wr}(\mathrm{r_0} \oplus 8)) \wedge \ldots}
$$

Figure 6: A Fragment of the formal safety proof of SP_r.

a way that the safety predicate is unchanged (for example, instruction scheduling and register allocation might do this in typical circumstances), or if both the code and the proof are modified so that we still have a valid proof of the new safety predicate, the validation succeeds and we continue to retain a guarantee of safety.

To automate the validation process, we must first choose a concrete representation language for predicates and their proofs. From the many available choices, we have selected the Edinburgh Logical Framework [7] (also called LF) as the representation framework for predicates and proofs. LF is an extension of the simply typed lambda calculus and was designed as a meta language for high-level specification of languages in logic and computer science. The most attractive property of LF is that it has a powerful yet simple typechecking algorithm, which we use to check the validity of proofs.

We represent the predicates and the proofs in LF in such a way that the validity of a proof is implied by the well typedness of the proof representation. Thus, proof validation amounts to typechecking. Also, LF allows us to represent in an elegant way a few key issues in logical proof correctness, such as the manipulation of logical parameters and assumptions. It is well beyond the scope of this paper to discuss in detail LF and the typechecking algorithm, however it is worth mentioning that typechecking is decidable and is described by a few simple rules. Indeed, typechecking is so simple that any programmers who do not trust the publicly available implementation can implement it easily themselves. Our implementation has about five pages of C code, even though it incorporates a few optimizations to the basic algorithm. With this implementation, it takes 1.4 milliseconds to validate the proof of the SP_r predicate.

For flexibility and to allow easy exchange of proofs between system components, we have designed a binary encoding of LF representations. Thus, a typical PCC binary contains a section with the native code ready to be mapped into memory and executed, followed by a symbol table used to reconstruct the LF representation at the code consumer site, and the binary encoding of the LF representation of the safety proof. Figure 7 shows the sizes of these sections for the PCC binary corresponding to the resource access example.

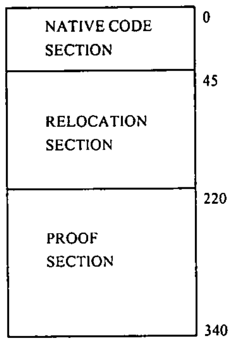

Figure 7: The layout of the PCC binary for the resource access example. The offsets are in bytes.

Currently, PCC binaries for standard packet filters, including the native code, safety proof, and relocation section, are about 400 to 1200 bytes in size, with the proof about 3 times larger than the code. The size of the relocation section increases linearly with the number of distinct proof rules used in the proof. In the case of packet filter safety proofs, the relocation section is a third of the binary but we expect this ratio be much smaller for larger proofs. There is a considerable amount of design latitude in the encodings of the proofs, and we have barely scratched the surface on what can be done to reduce the size of the binaries as well as the time required for validation. But already, with relatively little effort, we have achieved acceptably small binaries and low validation times.

3 Application: Network Packet Filters

In order to gain more experience with PCC and to compare it with other approaches to code safety, we have performed a series of experiments with safe network packet filters. We describe in this section the particulars of the PCC approach to network packet filters. Then in Section 3.1, we compare it with other approaches including interpreted packet filters (as exemplified by the BSD Packet Filter), code editing (through Software Fault Isolation), and using a safe programming language (the approach taken in the SPIN kernel).

A packet filter is an application-provided subroutine that scans each incoming network packet and decides whether the user application is interested in receiving it or not. Packet filters are supported by most of today's workstation operating systems. Since their first introduction in [15], packet filters have been used successfully in network monitoring and diagnosis.

In the PCC approach the packet filter is a PCC binary whose native code component is invoked by the kernel on each incoming network packet. Kernel safety is ensured by validating the safety proof.

Following the procedure described in Section 2 we first establish a safety policy. To allow for a fair comparison we follow the BSD Packet Filter model of safety. The packet filter code can examine the packet at will and can also write to a statically allocated scratch memory. Informally, the safety policy requires that: (1) memory reads are restricted to the packet and the scratch memory; (2) memory writes are limited to the scratch memory; (3) all branches are forward; and (4) reserved and callee-saves registers are not modified. These rules establish memory safety and termination assuming that the kernel calls the packet filter with valid packet and scratch memory addresses.

We write the packet filter code assuming that the return value must be in $\mathbf{r}_0$, the aligned address and the length of the packet filter are given in $\mathbf{r}_1$ and $\mathbf{r}_2$ respectively, and the address of a 16-byte aligned scratch memory is given in $\mathbf{r}_3$. Moreover the packet's length is positive and at least 64-bytes (the minimum length of an Ethernet packet). Formally this is expressed as the precondition:

$$\begin{array}{rl} Pre = & \mathbf{r}_1 \bmod 2^{64} = \mathbf{r}_1 \wedge \\ & \mathbf{r}_2 \bmod 2^{64} = \mathbf{r}_2 \wedge \mathbf{r}_2 < 2^{63} \wedge \mathbf{r}_2 \geq 64 \wedge \\ & \mathbf{r}_3 \bmod 2^{64} = \mathbf{r}_3 \wedge \\ & \forall i.(i \geq 0 \wedge i < \mathbf{r}_2 \wedge (i \,\&\, 7) = 0) \\ & \qquad \Rightarrow \mathbf{rd}(\mathbf{r}_1 \oplus i) \qquad \wedge \\ & \forall j.(j \geq 0 \wedge j < 16 \wedge (j \,\&\, 7) = 0) \\ & \qquad \Rightarrow \mathbf{wr}(\mathbf{r}_3 \oplus j) \qquad \wedge \\ & \forall i.\forall j.(i \geq 0 \wedge i < \mathbf{r}_2 \wedge j \geq 0 \wedge j < 16) \\ & \qquad \Rightarrow (\mathbf{r}_1 \oplus i \neq \mathbf{r}_3 \oplus j) \end{array}$$

The first few conjuncts of the precondition restrict the values of input registers to valid machine word values. The last term of the precondition rules out the possibility of memory aliasing between packets and the scratch memory. This is useful when reasoning about filters that write to the scratch memory.

The postcondition in our packet filter experiment is the predicate **true**, meaning that no additional conditions are placed on the final state.

We have implemented four typical packet filters in assembly language and certified their safety with respect to the packet filter safety policy. Filter 1 accepts all IP packets. This is done by comparing a 16-bit word in the packet to a given value. Filter 2 accepts IP packets originating from a given network. This involves checking a 24-bit value in addition to the work done by Filter 1. Filter 3 accepts IP or ARP packets exchanged between two given networks. This includes all the work done by Filter 2 with the addition of checking the destination network address. Extra complexity is required because of different header layout of IP and ARP packets. Filter 4 accepts all TCP packets with a given destination port. This filter has to check that the Ethernet packet is an IP packet, then that it is a TCP packet, and lastly that the destination port matches a given value. The offset of the TCP destination port is computed based on a byte in the IP header (the length of the IP header).

The effort involved in hand-coding packet filters in assembly language is repaid in increased performance, because packet filters are usually small and very frequently executed. Hand-coding provides the opportunity to perform optimizations that are difficult to obtain from an optimizing compiler. The important point is that these optimizations are not an impediment to generation and validation of safety proofs. Here are a few optimizations that we incorporated in our packet filters:

- The number of memory operations is minimized by using the DEC Alpha 64-bit load followed by byte extraction.

- The TCP port number can be found at packet offset $([14]_8 \,\&\, 15) * 4 + 16$, where $[14]_8$ denotes the byte at offset 14. If loading 64 bits at a time on a little-endian machine, the formula becomes $((([8]_{64} \gg 48) \,\&\, 255) \,\&\, 15) * 4 + 16$. With further simplification we reduce this to $(([8]_{64} \gg 46) \,\&\, 60) + 16$, which is exactly how we coded Filter 4.

After we write a packet filter, our prototype assembler produces its safety predicate using the verification-condition method presented in Section 2. The safety predicate is then proved using a theorem prover. We currently use our own theorem prover, which is admittedly a toy. When it gets stuck, it requires intervention from the programmer, mainly to learn new axioms about arithmetic (for example, to know that $\mathbf{r}_1 > 0 \Rightarrow \mathbf{r}_1 \geq 0$). The process is easy, and because user-provided axioms are remembered for future sessions, by now our system works automatically for most practical packet filters. With state-of-the-art theorem proving technology we expect to be able to prove completely automatically most arithmetic facts involved in certifying packet filters.

With our primitive theorem-prover we can generate safety proofs for packet filters in about 5 to 10 seconds, in the cases when no user-intervention is required.

3.1 Performance Comparisons

All performance measurements were done on a DEC Alpha 3000/600 with a 175-MHz processor, a 2-MByte secondary cache and 64-MByte main memory, running OSF/1. All measurements were performed off-line using a 200,000-packet trace from a busy Ethernet network at Carnegie Mellon University.

We measured the average per-packet run time of the four PCC packet filters and of functionally equivalent filters implemented using alternative approaches: the BSD Packet Filter architecture, Software Fault Isolation and programming in the safe subset of Modula-3. In our experiments with Modula-3 packet filters we use the VIEW extension [9] for pointer-safe casting. The result of the measurements are shown in Figure 8. From a per-packet latency point of view, the PCC packet filters outperform filters developed using any other considered approach. However, the PCC method has a startup cost significantly larger than the other approaches. This cost is the proof validation time, which is presented in Table 1 together with the PCC binary size for all four filters and maximum heap space used for validation. The maximum depth of the stack during validation was under 4 Kbytes.

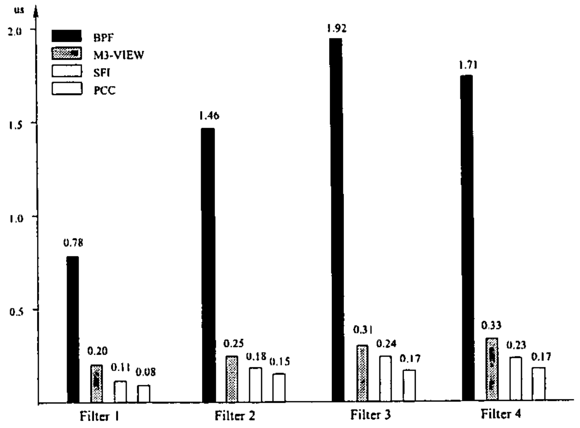

Figure 8: Comparison of average per-packet run time.

Packet Filter		1	2	3	4
Instructions		8	15	47	28
Binary Size (bytes)		385	516	1024	814
Validation Cost	Time (μs)	780	1070	2350	1710
	Space (KB)	5.5	8.7	24.6	15.1

Table 1: Proof size and validation cost for PCC packet filters.

Despite the relatively high validation cost, the run-time benefits of PCC packet filters are large enough to amortize the startup cost after processing a reasonable number of packets. Figure 9 shows the overall running time, including startup cost, as a function of the number of packets processed, for Filter 4. In this particular case, the cost of proof validation is amortized after 1200 packets when compared to the BPF version of the filter, after 10500 packets when compared to the Modula-3 version and after 28,000 packets when compared to the SFI packet filter. Note that at the time we collected the packet trace used for the experiments we counted about 1000 Ethernet packets per second on the average.

We proceed now to describe in more detail each considered approach focusing on how it relates to PCC from the safety point of view, and how we set up the performance measurements.

The standard way to ensure safe execution of packet filters is to interpret the filter and perform extensive run-time checks. This approach is best exemplified by the BSD Packet Filter architecture [13], commonly referred to as BPF. In the BPF approach the filter is encoded in a restricted accumulator-

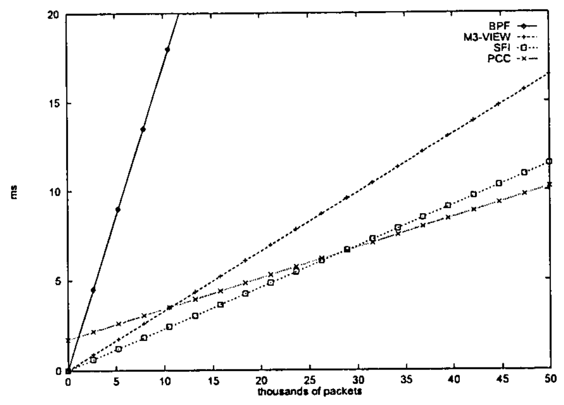

Figure 9: Startup cost amortization for Filter 4.

based language. According to the BPF semantics, a filter that attempts to read outside the packet or the scratch memory, or to write outside the scratch memory, is terminated and the packet rejected.

The BPF interpreter makes a simple static check of the packet filter code to verify that all instruction codes are valid and all branches are forward and within code limits. We measured this one-time overhead to be a few microseconds, which is negligible. BPF packet filters, however, are about 10 times slower than our PCC filters. In the PCC approach all checks are moved to the validation stage, allowing for much faster execution.

In order to collect data for the BPF packet filters, we extracted the BPF interpreter as implemented by the OSF/1 kernel and compiled it as a user library.

It is possible, of course, to eliminate the need for interpretation. For example, we could replace the packet-filter interpreter with a compiler. This approach is taken by several researchers [10, 24]. The problem here is the startup cost and complexity of compilation, especially if serious optimizations are performed.

Another approach to safe code execution is Software Fault Isolation (SFI) [23]. SFI is an inexpensive method for parsing binaries and inserting run-time checks on memory operations. There are many flavors of SFI depending on the desired level of memory safety. If the entire code runs in a single protection domain whose size is a power of 2, and if only memory writes are checked, then the run-time cost of SFI is relatively small. If, on the other hand, the untrusted code interacts frequently with the code consumer or other untrusted components residing in different protection domains and the read operations must be checked also, the overhead of the run-time checks can amount to 20% [23]. A more serious disadvantage of SFI is that it can only ensure memory safety. We believe that this level of safety is not enough in general, and that it is important to be able to check abstraction boundaries and representation invariants, as shown by the resource access example in Section 2.

In order to accommodate SFI for packet filters, we allowed some concessions to the packet filter semantics. For example, we assumed that the kernel allocates the packets on a 2048-byte boundary. Furthermore, we assume that the filter can safely access the entire segment of 2048 bytes, independently of the packet size. Note that the BPF packet filter semantics, which we followed for all other experiments, specifies that a filter should be terminated if it tries to access beyond the packet boundary. This means that some working packet filters in the BPF semantics will not behave as expected in the SFI semantics for packet filters, and vice-versa.

One common way of performing SFI is at the code producer site, usually as part of the code-generation phase in a compiler. In this case, the code consumer performs a load-time checking that SFI was done correctly. The load-time SFI validator is reportedly simple if it must deal only with binaries for which run-time checks have been inserted on every potentially dangerous memory operation [23]. On the other hand, in the case where the validator must accept binaries for which the number of run-time checks has been optimized through program analysis, the validator itself has to redo the analysis that led to the optimization. This means a more complex and slower validation, and in fact such an SFI validator does not presently exist.

We inserted run-time checks for the memory operations in the assembly language packet filters implemented for the PCC experiment. This process can be done by a simple and easy-to-trust implementation of SFI. In our experiments, PCC packet filters run about 25% faster than SFI filters.

As part of our SFI experiment, we produced safety proofs attesting that the resulting SFI packet filter binaries are safe with respect to the packet filter safety policy. We achieve the same effect as an SFI load-time validator but using the universal type-checking algorithm and a few application-dependent proof rules. The precondition for this experiment says that it is safe to read from any aligned address that is in the same 2048-byte segment with the packet start address. Proof sizes and validation times are very similar to those for plain PCC packets.

Another approach to safe code is to use a type-safe programming language. This approach is taken by the SPIN extensible operating system [1], and the language used is Modula-3 [17] extended with pointer-safe casting (VIEW). SPIN allows applications to install extensions in the kernel but only if they are written in the safe subset of Modula-3. The extensions are compiled by a trusted compiler and the resulting executable code is then believed to be safe (at least according to the Modula-3 model of safety). Note that such extensions written in Modula-3 are intrinsically safe, as anyone who believes in the safety of Modula-3 can check their compliance with Modula-3 syntactic and typing rules.

We believe that encoding kernel extensions as PCC binaries instead of Modula-3 source code can provide important benefits. One such benefit is the increased flexibility for extension writers because any native code extension can be accepted, independent of the original source language or even the compiler used, as long as a valid safety proof accompanies it. Another potential benefit is overcoming the limitations of the Modula-3 safety model: the PCC safety proof should be able to express properties such as disciplined use of locks or array bounds compliance with no need for run-time checks.

We wrote the four packet filters in the safe subset of Modula-3 and compiled them with the version 3.5 of the DEC SRC compiler extended with the VIEW operation [24]. VIEW is used to safely cast the packet filter to an array of aligned 64-bit words allowing fewer memory operation for accessing packet fields. In contrast, in plain Modula-3 the packet fields must be loaded a byte at a time, and a safety bounds check is performed for each such operation. The compiler tries to eliminate some of these checks statically but it is not very successful for packet filters. The main reason is that a critical piece of information, the fact that packets are at least 64 bytes long, cannot be communicated to the compiler through the Modula-3 type system.

We measured a 20% improvement in the Modula-3 packet filter performance when using VIEW. Similar performance improvements over the DEC SRC Modula-3 compiler have been reported [18] for the more recent Vortex compiler. However, since we have not conducted any experiments with the Vortex compiler on our packet filters, it is not clear what kind of improvements we would realize in practice.

In an alternate implementation of untrusted code certification using Modula-3, the source code is compiled by a trusted and secure compiler that signs the executable for future use. Validation then means cryptographic signature checking and like in the PCC approach there is no run-time cost associated with it. We do not have a complete implementation of such a cryptographic validation, so we do not know exactly how large is the startup cost for the digital signature approach. It is likely however that a good implementation of digital signatures would achieve faster validation and significantly faster generation of certificates. The essential drawback of cryptographic techniques over PCC is that validation establishes only a trusted origin of the code and not its absolute safety relative to the safety policy. In particular, a digital signature can be ascribed to an unsafe program just as easily as to a safe one. Also, the cost of managing and transmitting encryption keys is not incurred by PCC.

We should mention here one more approach to safe code execution, although we do not have an actual quantitative comparison. The Java Virtual Machine [21] is a proposed solution to safe interaction of distributed, untrusted agents. Mobile code is encoded in the Java Virtual Machine Language (also referred to as Java Bytecode), which is basically a safe low-level imperative language. Safety is achieved through a combination of static typechecking and run-time checking.

However, the Java Bytecode safety model is relatively limited as a result of limitations of the type system. For example the Java Bytecode type information encoded in the instruction codes can only express a few basic abstract types (e.g., integers, objects) and has no provisions for expressing safety policies like the one for the resource access example in Section 2. Also, invariants involving array bounds compliance cannot be expressed in the Java Bytecode type system and must be checked at run time.

Although Java Bytecode is a low-level language, it still requires substantial processing before it can be executed on a general-purpose processor. In contrast, PCC segregates the safety proof from the program code, allowing for the code portion to be encoded in a variety of languages, including native code, without any safety loss.

4 Practical Problems and Future Work

In order to create a safety proof, the code producer must prove a predicate in first-order logic. In general, this problem is undecidable. However, as we mentioned in Section 1, the code producer can resort to "extra" run-time checks inserted in strategic locations, which have the tendency to simplify the certification.

Fortunately, in the packet-filter experiments, the certification process is nearly automatic, and we have not been forced to insert any extra run-time checks into the code. In fact, we find that safety predicates for packet filters are fairly easy handled by existing theorem-proving technology.

One of the simplifications in the packet filters is to restrict programs so that they do not contain loops. Although the general framework presented in this paper is easily extended to accommodate loops [5], this introduces a number of complications. One experiment we conducted involves an IP-header checksum routine, which is hand-coded in 39 DEC Alpha instructions. The core loop contains 8 instructions, and is optimized by computing the 16-bit IP checksum using 64-bit additions followed by a folding operation. The resulting PCC binary for this routine is, as expected, quite fast, beating the standard C version in the OSF/1 kernel by a factor of two. The PCC binary itself is 1610 bytes in size and proof validation takes 3.6 milliseconds.

This experiment brought to light several complications. For example, the standard approach of verifying loops using Floyd-style verification conditions involves introducing loop invariants explicitly, which is a challenge for any theorem-proving technology and ofter requires user intervention. In fact, for general assembly-language programs this represents the most important problem to be solved, as it is the main obstacle in automating the generation of proofs. Since this is beyond the capabilities of our system, we are forced to write the invariants out by hand. This also means that the native code must be accompanied by a loop invariant for every loop. Thus, the PCC binary contains a mapping between each loop and its invariant. Our convention is to have the PCC binary contain a table that maps each backward-branch target to a loop invariant.

Besides the problem of how to generate the proofs, there is also the matter of their size. In principle, the proofs can be exponentially large (in the size of the program). This has not been a problem for any of the examples we have tried thus far, however. The blowup would tend to occur in programs that contain long sequences of conditionals, with no intervening loops. Perhaps we have not yet seen the problem in a serious way because such programs tend to be hard for humans to understand, and we are writing the programs by hand. But as a general matter, the size of the PCC binaries is an issue that must be addressed carefully. We have implemented several optimizations in the representation of the proofs, and much more is possible here. But ultimately, we need more practical experience to know if this is a serious obstacle for PCC in practice.

For programs with loops, the loop invariants break a program with cycles into a set of acyclic code fragments. We treat each code fragment as a separate program, using the invariants as preconditions for each. This has the beneficial effect of partitioning the safety predicate and its proof into smaller pieces, and overall tends to reduce the size of the proof dramatically. For this reason, even for sections of programs that do not contain loops, it may be beneficial to introduce invariants, as a way of controlling the growth of the PCC binaries.

In addition to developing better certification technology, we see several other interesting directions for further research. One possibility that we intend to explore is the application of PCC to more dynamic properties, such as resource-usage guarantees. One example would be to certify that specific synchronization locks are always released prior to some action. The framework we have presented in this paper is already expressive enough to define such safety policies, and so what remains now is to try some experiments.

Another possibility is to allow the consumer and producer to "negotiate" a safety policy at run time. This would work by allowing the producer to send an encoding of a proposed safety policy (including the VC-generation rules, proof rules, and preconditions) to the consumer. If the consumer determines that the proposed policy implies some basic notion of safety, then it can allow the producer to produce PCC binaries using the new policy. This might be useful in distributed systems, in which one agent wants to define a language and then transmit to other agents code written in that language.

Finally, we believe there would be advantages to starting with a safe programming language and then implementing a *certifying compiler* that produces PCC binaries as target programs. For the safety properties that are implied by the source language, construction of the proofs is, in principle, a matter of having the compiler prove the correctness of the translation to target code. We have already experimented with a toy compiler of this sort for a small type-safe programming language, and hope to expand on this in the near future.

5 Conclusions

We have described *proof-carrying code,* a mechanism that allows a kernel or server to interact safely with binaries supplied by an untrusted source. PCC does not incur any run-time overhead for the kernel. Instead, the code producer is required to generate a formal proof that the code obeys the safety policy.

The kernel can easily check the proofs for validity, after which it is absolutely certain that the code respects the safety policy. Furthermore, PCC binaries are completely tamper-proof; any attempt to alter either the native code or safety proof in a PCC binary is either detected or harmless. Our experiments with network packet filters show that PCC can lead to significant performance advantages over existing approaches to safe code, including code-editing techniques such as Software Fault Isolation.

Proof-carrying code has the potential to free the system designer from relying on run-time checking as the sole means of ensuring safety. Traditionally, system designers have always viewed safety simply in terms of memory protection, achieved through the use of rather expensive run-time mechanisms such as hardware-enforced memory protection and extensive run-time checking of data. By being limited to memory protection and run-time checking, the designer must impose substantial restrictions on the structure and implementation of the entire system, for example by requiring the use of a restricted application-kernel interaction model (such as a fixed system call or application-program interface.)

Proof-carrying code, on the other hand, allows the safety policy to be defined by the kernel designer and then certified by each application. Not only does this provide greater flexibility for designers of both the system and applications, but also allows safety policies to be used that are more abstract and fine-grained than memory protection. We believe that this has the potential to lead to great improvements in the robustness and end-to-end performance of systems.

6 Final Thoughts

The inspiration for proof-carrying code comes from the realm of static type systems, especially as embodied by the language Standard ML (SML). In the formal definition of SML [14], a formal theorem guarantees the safety of any type-correct SML program, for a rigorously defined notion of safety. There are, of course, many other type-safe programming languages, for example Modula-3 [17] and Java [20], but the use of mathematical formalism sets SML apart from the these languages, and as a practical matter this rigor provides the basic conceptual and technical foundations that we need to create checkable proofs.

With type-safe languages like SML in mind, we can get an intuitive idea about how proof-carrying code works. Consider a compiler for SML. Agent A writes an SML program and compiles it to a native-code target program. If we then throw away the source program, how can we later convince an agent B that the target program is safe? (We are assuming that agent B does not trust agent A.) One way to do this is to have the compiler *prove* that the target code correctly corresponds to the source code.[4] Now, as it turns out, in the type theory of SML, not only can such a proof be written out formally, but in fact it can be written in a typed language with the property that any well-typed proof is guaranteed to be valid.

Proof-carrying code is thus an application of ideas from programming-language theory, in this case used for defining notions of safety that are useful for operating systems, and flexible enough to accommodate both high-level and low-level languages. With the growth of interest in highly distributed computing, web computing, and extensible kernels, it seems clear to us that ideas from programming languages are destined to become increasingly critical for robust and good-performing systems.

7 Acknowledgements

We thank Robert Harper, Brian Noble, Daniel Jackson, Edo Biagioni, Greg Morrisett, Scott Draves, Chris Colby, Martin Abadi and Dave Detlefs for reading previous versions of this paper and for suggesting many improvements. We also thank Charles Garrett, Brian Bershad, Wilson Hsieh for suggesting many improvements to the methodology for the Modula-3 performance measurements. Finally, we thank the anonymous reviewers for their many suggestions for improving this paper. In particular we thank our shepherd, Jay Lepreau, who also suggested the PCC name.

References

[1] BERSHAD, B., SAVAGE, S., PARDYAK, P., SIRER, E. G., BECKER, D., FIUCZYNSKI, M., CHAMBERS, C., AND EGGERS, S. Extensibility, safety and performance in the SPIN operating system. In *Symposium on Operating System Principles* (Dec. 1995), pp. 267–284.

[2] BOYER, R. S., AND YU, Y. Automated proofs of object code for a widely used microprocessor. *J. ACM 43*, 1 (Jan. 1996), 166–192.

[4] This is essentially the same as having a compiler translate the types as well as the code, so that the target program will have types that can be checked. In fact, this approach to compiling is taken by the SML/TIL compiler [22].

[3] Clutterbuck, D., and Carré, B. The verification of low-level code. *IEEE Software Engineering Journal 3*, 3 (May 1988), 97–111.

[4] Constable, R. L., Allen, S. F., Bromley, H. M., Cleaveland, W. R., Cremer, J. F., Harper, R. W., Howe, D. J., Knoblock, T. B., Mendler, N. P., Panangaden, P., Sasaki, J. T., and Smith, S. F. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall, 1986.

[5] Dijkstra, E. W. Guarded commands, nondeterminancy and formal derivation of programs. *Communications of the ACM 18* (1975), 453–457.

[6] Floyd, R. W. Assigning meanings to programs. In *Mathematical Aspects of Computer Science*, J. T. Schwartz, Ed. American Mathematical Society, 1967, pp. 19–32.

[7] Harper, R., Honsell, F., and Plotkin, G. A framework for defining logics. *Journal of the Association for Computing Machinery 40*, 1 (Jan. 1993), 143–184.

[8] Hoare, C. A. R. An axiomatic basis for computer programming. *Communications of the ACM 12* (1969), 567–580.

[9] Hsieh, W. C., Fiuczynski, M. E., Garrett, C., Savage, S., Becker, D., and Bershad, B. N. Language support for extensible operating systems. In *The Inaugural Workshop on Compiler Support for Systems Software* (Feb. 1996), pp. 127–133.

[10] Lee, P., and Leone, M. Optimizing ML with run-time code generation. In *PLDI'96 Conference on Programming Language Design and Implementation* (May 1996), pp. 137–148.

[11] Martin-Löf, P. A theory of types. Technical Report 71-3, Department of Mathematics, University of Stockholm, 1971.

[12] McCanne, S. The Berkeley Packet Filter man page. BPF distribution available at `ftp://ftp.ee.lbl.gov`, May 1991.

[13] McCanne, S., and Jacobson, V. The BSD packet filter: A new architecture for user-level packet capture. In *The Winter 1993 USENIX Conference* (Jan. 1993), USENIX Association, pp. 259–269.

[14] Milner, R., Tofte, M., and Harper, R. *The Definition of Standard ML.* MIT Press, Cambridge, Massachusetts, 1990.

[15] Mogul, J. C., Rashid, R. F., and Accetta, M. J. The packet filter: An efficient mechanism for user-level network code. In *ACM Symposium on Operating Systems Principles* (Nov. 1987), ACM Press, pp. 39–51. An updated version is available as DEC WRL Research Report 87/2.

[16] Necula, G. C., and Lee, P. Proof-carrying code. Technical Report CMU-CS-96-165, Computer Science Department, Carnegie Mellon University, Dec. 1996. Also appeared as FOX memorandum CMU-CS-FOX-96-03.

[17] Nelson, G. *Systems Programming with MODULA-3.* Prentice-Hall, 1991.

[18] Sirer, E. G., Savage, S., Pardyak, P., DeFouw, G. P., and Bershad, B. N. Writing an operating system with Modula-3. In *The Inaugural Workshop on Compiler Support for Systems Software* (Feb. 1996), pp. 134–140.

[19] Sites, R. L. *Alpha Architecture Reference Manual.* Digital Press, 1992.

[20] Sun Microsystems. The Java language specification. Available as `ftp://ftp.javasoft.com/docs/javaspec.ps.zip`, 1995.

[21] Sun Microsystems. The Java Virtual Machine specification. Available as `ftp://ftp.javasoft.com/docs/vmspec.ps.zip`, 1995.

[22] Tarditi, D., Morrisett, J. G., Cheng, P., Stone, C., Harper, R., and Lee, P. TIL: A type-directed optimizing compiler for ML. In *PLDI'96 Conference on Programming Language Design and Implementation* (May 1996), pp. 181–192.

[23] Wahbe, R., Lucco, S., Anderson, T. E., and Graham, S. L. Efficient software-based fault isolation. In *14th ACM Symposium on Operating Systems Principles* (Dec. 1993), ACM, pp. 203–216.

[24] Wallach, D. A., Engler, D., and Kaashoek, M. F. ASHs : Application-specific handlers for high-performance messaging. In *ACM SIGCOMM'96* (Oct. 1996), vol. 26, ACM.

Efficient Software-Based Fault Isolation

Robert Wahbe Steven Lucco Thomas E. Anderson Susan L. Graham

Computer Science Division
University of California
Berkeley, CA 94720

Abstract

One way to provide fault isolation among cooperating software modules is to place each in its own address space. However, for tightly-coupled modules, this solution incurs prohibitive context switch overhead. In this paper, we present a software approach to implementing fault isolation within a single address space. Our approach has two parts. First, we load the code and data for a distrusted module into its own *fault domain*, a logically separate portion of the application's address space. Second, we modify the object code of a distrusted module to prevent it from writing or jumping to an address outside its fault domain. Both these software operations are portable and programming language independent.

Our approach poses a tradeoff relative to hardware fault isolation: substantially faster communication between fault domains, at a cost of slightly increased execution time for distrusted modules. We demonstrate that for frequently communicating modules, implementing fault isolation in software rather than hardware can substantially improve end-to-end application performance.

This work was supported in part by the National Science Foundation (CDA-8722788), Defense Advanced Research Projects Agency (DARPA) under grant MDA972-92-J-1028 and contracts DABT63-92-C-0026 and N00600-93-C-2481, the Digital Equipment Corporation (the Systems Research Center and the External Research Program), and the AT&T Foundation. Anderson was also supported by a National Science Foundation Young Investigator Award. The content of the paper does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

Email: {rwahbe, lucco, tea, graham}@cs.berkeley.edu

1 Introduction

Application programs often achieve extensibility by incorporating independently developed software modules. However, faults in extension code can render a software system unreliable, or even dangerous, since such faults could corrupt permanent data. To increase the reliability of these applications, an operating system can provide services that prevent faults in distrusted modules from corrupting application data. Such *fault isolation* services also facilitate software development by helping to identify sources of system failure.

For example, the POSTGRES database manager includes an extensible type system [Sto87]. Using this facility, POSTGRES queries can refer to general-purpose code that defines constructors, destructors, and predicates for user-defined data types such as geometric objects. Without fault isolation, any query that uses extension code could interfere with an unrelated query or corrupt the database.

Similarly, recent operating system research has focused on making it easier for third party vendors to enhance parts of the operating system. An example is micro-kernel design; parts of the operating system are implemented as user-level servers that can be easily modified or replaced. More generally, several systems have added extension code into the operating system, for example, the BSD network packet filter [MRA87, MJ93], application-specific virtual memory management [HC92], and Active Messages [vCGS92]. Among industry systems, Microsoft's Object Linking and Embedding system [Cla92] can link together independently developed software modules. Also, the Quark Xpress desktop publishing system [Dys92] is structured to support incorporation of general-purpose third party code. As with POSTGRES, faults in extension modules can render any of these systems unreliable.

One way to provide fault isolation among cooperating software modules is to place each in its own address space. Using Remote Procedure Call (RPC) [BN84], modules in separate address spaces can call into each other through a normal procedure call interface. Hardware page tables prevent the code in one address space from corrupting the contents of another.

Unfortunately, there is a high performance cost to providing fault isolation through separate address spaces. Transferring control across protection boundaries is expensive, and does not necessarily scale with improvements in a processor's integer performance [ALBL91]. A cross-address-space RPC requires at least: a trap into the operating system kernel, copying each argument from the caller to the callee, saving and restoring registers, switching hardware address spaces (on many machines, flushing the translation lookaside buffer), and a trap back to user level. These operations must be repeated upon RPC return. The execution time overhead of an RPC, even with a highly optimized implementation, will often be two to three orders of magnitude greater than the execution time overhead of a normal procedure call [BALL90, ALBL91].

The goal of our work is to make fault isolation cheap enough that system developers can ignore its performance effect in choosing which modules to place in separate fault domains. In many cases where fault isolation would be useful, cross-domain procedure calls are frequent yet involve only a moderate amount of computation per call. In this situation it is impractical to isolate each logically separate module within its own address space, because of the cost of crossing hardware protection boundaries.

We propose a software approach to implementing fault isolation within a single address space. Our approach has two parts. First, we load the code and data for a distrusted module into its own *fault domain*, a logically separate portion of the application's address space. A fault domain, in addition to comprising a contiguous region of memory within an address space, has a unique identifier which is used to control its access to process resources such as file descriptors. Second, we modify the object code of a distrusted module to prevent it from writing or jumping to an address outside its fault domain. Program modules isolated in separate software-enforced fault domains can not modify each other's data or execute each other's code except through an explicit cross-fault-domain RPC interface.

We have identified several programming-language-independent transformation strategies that can render object code unable to escape its own code and data segments. In this paper, we concentrate on a simple transformation technique, called *sandboxing*, that only slightly increases the execution time of the modified object code. We also investigate techniques that provide more debugging information but which incur greater execution time overhead.

Our approach poses a tradeoff relative to hardware-based fault isolation. Because we eliminate the need to cross hardware boundaries, we can offer substantially lower-cost RPC between fault domains. A safe RPC in our prototype implementation takes roughly $1.1\mu s$ on a DECstation 5000/240 and roughly $0.8\mu s$ on a DEC Alpha 400, more than an order of magnitude faster than any existing RPC system. This reduction in RPC time comes at a cost of slightly increased distrusted module execution time. On a test suite including the the C SPEC92 benchmarks, sandboxing incurs an average of 4% execution time overhead on both the DECstation and the Alpha.

Software-enforced fault isolation may seem to be counter-intuitive: we are slowing down the common case (normal execution) to speed up the uncommon case (cross-domain communication). But for frequently communicating fault domains, our approach can offer substantially better end-to-end performance. To demonstrate this, we applied software-enforced fault isolation to the POSTGRES database system running the Sequoia 2000 benchmark. The benchmark makes use of the POSTGRES extensible data type system to define geometric operators. For this benchmark, the software approach reduced fault isolation overhead by more than a factor of three on a DECstation 5000/240.

A software approach also provides a tradeoff between performance and level of distrust. If some modules in a program are trusted while others are distrusted (as may be the case with extension code), only the distrusted modules incur any execution time overhead. Code in trusted domains can run at full speed. Similarly, it is possible to use our techniques to implement full security, preventing distrusted code from even *reading* data outside of its domain, at a cost of higher execution time overhead. We quantify this effect in Section 5.

The remainder of the paper is organized as follows. Section 2 provides some examples of systems that require frequent communication between fault domains. Section 3 outlines how we modify object code to prevent it from generating illegal addresses. Section 4 describes how we implement low latency cross-fault-domain RPC. Section 5 presents performance results for our prototype, and finally Section 6 discusses some related work.

2 Background

In this section, we characterize in more detail the type of application that can benefit from software-enforced fault isolation. We defer further description of the POSTGRES extensible type system until Section 5, which gives performance measurements for this application.

The operating systems community has focused considerable attention on supporting kernel extensibility. For example, the UNIX vnode interface is designed to make it easy to add a new file system into UNIX [Kle86]. Unfortunately, it is too expensive to forward every file system operation to user level, so typically new file system implementations are added directly into the kernel. (The Andrew file system is largely implemented at user level, but it maintains a kernel cache for performance [HKM+88].) Epoch's tertiary storage file system [Web93] is one example of operating system kernel code developed by a third party vendor.

Another example is user-programmable high performance I/O systems. If data is arriving on an I/O channel at a high enough rate, performance will be degraded substantially if control has to be transferred to user level to manipulate the incoming data [FP93]. Similarly, Active Messages provide high performance message handling in distributed-memory multiprocessors [vCGS92]. Typically, the message handlers are application-specific, but unless the network controller can be accessed from user level [Thi92], the message handlers must be compiled into the kernel for reasonable performance.

A user-level example is the Quark Xpress desktop publishing system. One can purchase third party software that will extend this system to perform functions unforeseen by its original designers [Dys92]. At the same time, this extensibility has caused Quark a number of problems. Because of the lack of efficient fault domains on the personal computers where Quark Xpress runs, extension modules can corrupt Quark's internal data structures. Hence, bugs in third party code can make the Quark system appear unreliable, because end-users do not distinguish among sources of system failure.

All these examples share two characteristics. First, using hardware fault isolation would result in a significant portion of the overall execution time being spent in operating system context switch code. Second, only a small amount of code is distrusted; most of the execution time is spent in trusted code. In this situation, software fault isolation is likely to be more efficient than hardware fault isolation because it sharply reduces the time spent crossing fault domain boundaries, while only slightly increasing the time spent executing the distrusted part of the application. Section 5 quantifies this trade-off between domain-crossing overhead and application execution time overhead, and demonstrates that even if domain-crossing overhead represents a modest proportion of the total application execution time, software-enforced fault isolation is cost effective.

3 Software-Enforced Fault Isolation

In this section, we outline several *software encapsulation* techniques for transforming a distrusted module so that it can not escape its fault domain. We first describe a technique that allows users to pinpoint the location of faults within a software module. Next, we introduce a technique, called *sandboxing*, that can isolate a distrusted module while only slightly increasing its execution time. Section 5 provides a performance analysis of this techinique. Finally, we present a software encapsulation technique that allows cooperating fault domains to share memory. The remainder of this discussion assumes we are operating on a RISC load/store architecture, although our techniques could be extended to handle CISCs. Section 4 describes how we implement safe and efficient cross-fault-domain RPC.

We divide an application's virtual address space into segments, aligned so that all virtual addresses within a segment share a unique pattern of upper bits, called the *segment identifier*. A fault domain consists of two segments, one for a distrusted module's code, the other for its static data, heap and stack. The specific segment addresses are determined at load time.

Software encapsulation transforms a distrusted module's object code so that it can jump only to targets in its code segment, and write only to addresses within its data segment. Hence, all legal jump targets in the distrusted module have the same upper bit pattern (segment identifier); similarly, all legal data addresses generated by the distrusted module share the same segment identifier. Separate code and data segments are necessary to prevent a module from modifying its code segment[1]. It is possible for an address with the correct segment identifier to be illegal, for instance if it refers to an unmapped page. This is caught by the normal operating system page fault mechanism.

3.1 Segment Matching

An *unsafe instruction* is any instruction that jumps to or stores to an address that can not be statically ver-

[1]Our system supports dynamic linking through a special interface.

ified to be within the correct segment. Most control transfer instructions, such as program-counter-relative branches, can be statically verified. Stores to static variables often use an immediate addressing mode and can be statically verified. However, jumps through registers, most commonly used to implement procedure returns, and stores that use a register to hold their target address, can not be statically verified.

A straightforward approach to preventing the use of illegal addresses is to insert checking code before every unsafe instruction. The checking code determines whether the unsafe instruction's target address has the correct segment identifier. If the check fails, the inserted code will trap to a system error routine outside the distrusted module's fault domain. We call this software encapsulation technique *segment matching*.

On typical RISC architectures, segment matching requires four instructions. Figure 1 lists a pseudo-code fragment for segment matching. The first instruction in this fragment moves the store target address into a *dedicated register*. Dedicated registers are used only by inserted code and are never modified by code in the distrusted module. They are necessary because code elsewhere in the distrusted module may arrange to jump directly to the unsafe store instruction, bypassing the inserted check. Hence, we transform all unsafe store and jump instructions to use a dedicated register.

All the software encapsulation techniques presented in this paper require dedicated registers[2]. Segment matching requires four dedicated registers: one to hold addresses in the code segment, one to hold addresses in the data segment, one to hold the segment shift amount, and one to hold the segment identifier.

Using dedicated registers may have an impact on the execution time of the distrusted module. However, since most modern RISC architectures, including the MIPS and Alpha, have at least 32 registers, we can retarget the compiler to use a smaller register set with minimal performance impact. For example, Section 5 shows that, on the DECstation 5000/240, reducing by five registers the register set available to a C compiler (gcc) did not have a significant effect on the average execution time of the SPEC92 benchmarks.

3.2 Address Sandboxing

The segment matching technique has the advantage that it can pinpoint the offending instruction. This capability is useful during software development. We can reduce runtime overhead still further, at the cost of providing no information about the source of faults.

[2]For architectures with limited register sets, such as the 80386 [Int86], it is possible to encapsulate a module using no reserved registers by restricting control flow within a fault domain.

```
dedicated-reg ⇐ target address
    Move target address into dedicated register.
scratch-reg ⇐ (dedicated-reg>>shift-reg)
    Right-shift address to get segment identifier.
    scratch-reg is not a dedicated register.
    shift-reg is a dedicated register.
compare scratch-reg and segment-reg
    segment-reg is a dedicated register.
trap if not equal
    Trap if store address is outside of segment.
store instruction uses dedicated-reg
```

Figure 1: Assembly pseudo code for segment matching.

```
dedicated-reg ⇐ target-reg&and-mask-reg
    Use dedicated register and-mask-reg
    to clear segment identifier bits.
dedicated-reg ⇐ dedicated-reg|segment-reg
    Use dedicated register segment-reg
    to set segment identifier bits.
store instruction uses dedicated-reg
```

Figure 2: Assembly pseudo code to sandbox address in `target-reg`.

Before each unsafe instruction we simply insert code that *sets* the upper bits of the target address to the correct segment identifier. We call this *sandboxing* the address. Sandboxing does not catch illegal addresses; it merely prevents them from affecting any fault domain other than the one generating the address.

Address sandboxing requires insertion of two arithmetic instructions before each unsafe store or jump instruction. The first inserted instruction clears the segment identifier bits and stores the result in a dedicated register. The second instruction sets the segment identifier to the correct value. Figure 2 lists the pseudo-code to perform this operation. As with segment matching, we modify the unsafe store or jump instruction to use the dedicated register. Since we are using a dedicated register, the distrusted module code can not produce an illegal address even by jumping to the second instruction in the sandboxing sequence; since the upper bits of the dedicated register will already contain the correct segment identifier, this second instruction will have no effect. Section 3.6 presents a simple algorithm that can verify that an object code module has been correctly sandboxed.

Address sandboxing requires five dedicated registers. One register is used to hold the segment mask, two registers are used to hold the code and data segment

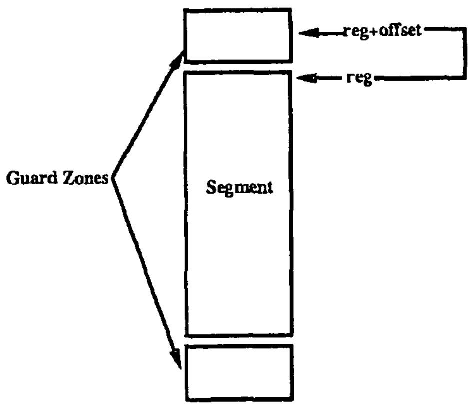

Figure 3: A segment with guard zones. The size of the guard zones covers the range of possible immediate offsets in register-plus-offset addressing modes.

identifiers, and two are used to hold the sandboxed code and data addresses.

3.3 Optimizations

The overhead of software encapsulation can be reduced by using conventional compiler optimizations. Our current prototype applies loop invariant code motion and instruction scheduling optimizations [ASU86, ACD74]. In addition to these conventional techniques, we employ a number of optimizations specialized to software encapsulation.

We can reduce the overhead of software encapsulation mechanisms by avoiding arithmetic that computes target addresses. For example, many RISC architectures include a register-plus-offset instruction mode, where the offset is an immediate constant in some limited range. On the MIPS architecture such offsets are limited to the range -64K to +64K. Consider the store instruction **`store value,offset(reg)`**, whose address **`offset(reg)`** uses the register-plus-offset addressing mode. Sandboxing this instruction requires three inserted instructions: one to sum **`reg+offset`** into the dedicated register, and two sandboxing instructions to set the segment identifier of the dedicated register.

Our prototype optimizes this case by sandboxing only the register **`reg`**, rather than the actual target address **`reg+offset`**, thereby saving an instruction. To support this optimization, the prototype establishes *guard zones* at the top and bottom of each segment. To create the guard zones, virtual memory pages adjacent to the segment are unmapped (see Figure 3).

We also reduce runtime overhead by treating the MIPS stack pointer as a dedicated register. We avoid sandboxing the uses of the stack pointer by sandboxing this register whenever it is set. Since uses of the stack pointer to form addresses are much more plentiful than changes to it, this optimization significantly improves performance.

Further, we can avoid sandboxing the stack pointer after it is modified by a small constant offset as long as the modified stack pointer is used as part of a load or store address before the next control transfer instruction. If the modified stack pointer has moved into a guard zone, the load or store instruction using it will cause a hardware address fault. On the DEC Alpha processor, we apply these optimizations to both the frame pointer and the stack pointer.

There are a number of further optimizations that could reduce sandboxing overhead. For example, the transformation tool could remove sandboxing sequences from loops, in cases where a store target address changes by only a small constant offset during each loop iteration. Our prototype does not yet implement these optimizations.

3.4 Process Resources

Because multiple fault domains share the same virtual address space, the fault domain implementation must prevent distrusted modules from corrupting resources that are allocated on a per-address-space basis. For example, if a fault domain is allowed to make system calls, it can close or delete files needed by other code executing in the address space, potentially causing the application as a whole to crash.

One solution is to modify the operating system to know about fault domains. On a system call or page fault, the kernel can use the program counter to determine the currently executing fault domain, and restrict resources accordingly.

To keep our prototype portable, we implemented an alternative approach. In addition to placing each distrusted module in a separate fault domain, we require distrusted modules to access system resources only through cross-fault-domain RPC. We reserve a fault domain to hold trusted *arbitration* code that determines whether a particular system call performed by some other fault domain is safe. If a distrusted module's object code performs a direct system call, we transform this call into the appropriate RPC call. In the case of an extensible application, the trusted portion of the application can make system calls directly and shares a fault domain with the arbitration code.

3.5 Data Sharing

Hardware fault isolation mechanisms can support data sharing among virtual address spaces by manipulating page table entries. Fault domains share an address space, and hence a set of page table entries, so they can not use a standard shared memory implementation. Read-only sharing is straightforward; since our software encapsulation techniques do not alter load instructions, fault domains can read any memory mapped in the application's address space [3].

If the object code in a particular distrusted module has been sandboxed, then it can share read-write memory with other fault domains through a technique we call *lazy pointer swizzling*. Lazy pointer swizzling provides a mechanism for fault domains to share arbitrarily many read-write memory regions with no additional runtime overhead. To support this technique, we modify the hardware page tables to map the shared memory region into every address space segment that needs access; the region is mapped at the same offset in each segment. In other words, we alias the shared region into multiple locations in the virtual address space, but each aliased location has exactly the same low order address bits. As with hardware shared memory schemes, each shared region must have a different segment offset.

To avoid incorrect shared pointer comparisons in sandboxed code, the shared memory creation interface must ensure that each shared object is given a unique address. As the distrusted object code accesses shared memory, the sandboxing code automatically translates shared addresses into the corresponding addresses within the fault domain's data segment. This translation works exactly like hardware translation; the low bits of the address remain the same, and the high bits are set to the data segment identifier.

Under operating systems that do not allow virtual address aliasing, we can implement shared regions by introducing a new software encapsulation technique: *shared segment matching*. To implement sharing, we use a dedicated register to hold a bitmap. The bitmap indicates which segments the fault domain can access. For each unsafe instruction checked, shared segment matching requires one more instruction than segment matching.

3.6 Implementation and Verification

We have identified two strategies for implementing software encapsulation. One approach uses a compiler to emit encapsulated object code for a distrusted module; the integrity of this code is then verified when the module is loaded into a fault domain. Alternatively, the system can encapsulate the distrusted module by directly modifying its object code at load time.

[3] We have implemented versions of these techniques that perform general protection by encapsulating load instructions as well as store and jump instructions. We discuss the performance of these variants in Section 5.

Our current prototype uses the first approach. We modified a version of the gcc compiler to perform software encapsulation. Note that while our current implementation is language dependent, our techniques are language independent.

We built a verifier for the MIPS instruction set that works for both sandboxing and segment matching. The main challenge in verification is that, in the presence of indirect jumps, execution may begin on any instruction in the code segment. To address this situation, the verifier uses a property of our software encapsulation techniques: all unsafe stores and jumps use a dedicated register to form their target address. The verifier divides the program into sequences of instructions called *unsafe regions*. An *unsafe store region* begins with any modification to a dedicated store register. An *unsafe jump region* begins with any modification to a dedicated jump register. If the first instruction in a unsafe store or jump region is executed, all subsequent instructions are guaranteed to be executed. An unsafe store region ends when one of the following hold: the next instruction is a store which uses a dedicated register to form its target address, the next instruction is a control transfer instruction, the next instruction is not guaranteed to be executed, or there are no more instructions in the code segment. A similar definition is used for unsafe jump regions.

The verifier analyzes each unsafe store or jump region to insure that any dedicated register modified in the region is valid upon exit of the region. For example, a load to a dedicated register begins an unsafe region. If the region appropriately sandboxes the dedicated register, the unsafe region is deemed safe. If an unsafe region can not be verified, the code is rejected.

By incorporating software encapsulation into an existing compiler, we are able to take advantage of compiler infrastructure for code optimization. However, this approach has two disadvantages. First, most modified compilers will support only one programming language (gcc supports C, C++, and Pascal). Second, the compiler and verifier must be synchronized with respect to the particular encapsulation technique being employed.

An alternative, called *binary patching*, alleviates these problems. When the fault domain is loaded, the system can encapsulate the module by directly modifying the object code. Unfortunately, practical and robust binary patching, resulting in efficient code, is not currently possible [LB92]. Tools which translate one binary format to another have been built, but these tools rely on compiler-specific idioms to distinguish code from data and use processor emulation to handle unknown indirect jumps[SCK+93]. For software encapsulation, the main challenge is to transform the code so that it uses a subset of the registers, leav-

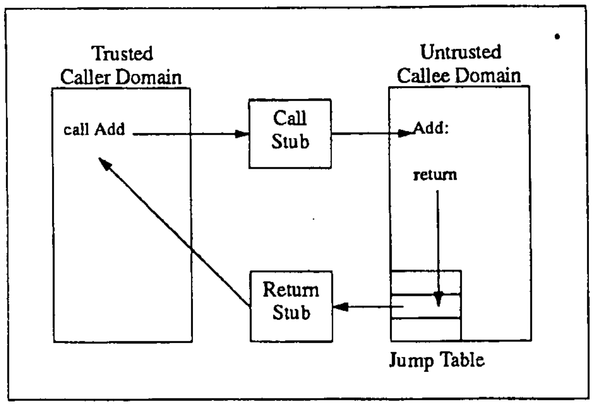

Figure 4: Major components of a cross-fault-domain RPC.

ing registers available for dedicated use. To solve this problem, we are working on a binary patching prototype that uses simple extensions to current object file formats. The extensions store control flow and register usage information that is sufficient to support software encapsulation.

4 Low Latency Cross Fault Domain Communication

The purpose of this work is to reduce the cost of fault isolation for cooperating but distrustful software modules. In the last section, we presented one half of our solution: efficient software encapsulation. In this section, we describe the other half: fast communication across fault domains.

Figure 4 illustrates the major components of a cross-fault-domain RPC between a trusted and distrusted fault domain. This section concentrates on three aspects of fault domain crossing. First, we describe a simple mechanism which allows a fault domain to safely call a trusted stub routine outside its domain; that stub routine then safely calls into the destination domain. Second, we discuss how arguments are efficiently passed among fault domains. Third, we detail how registers and other machine state are managed on cross-fault-domain RPCs to insure fault isolation. The protocol for exporting and naming procedures among fault domains is independent of our techniques.

The only way for control to escape a fault domain is via a *jump table*. Each jump table entry is a control transfer instruction whose target address is a legal entry point outside the domain. By using instructions whose target address is an immediate encoded in the instruction, the jump table does not rely on the use of a dedicated register. Because the table is kept in the (read-only) code segment, it can only be modified by a trusted module.

For each pair of fault domains a customized call and return stub is created for each exported procedure. Currently, the stubs are generated by hand rather than using a stub generator [JRT85]. The stubs run unprotected outside of both the caller and callee domain. The stubs are responsible for copying cross-domain arguments between domains and managing machine state.

Because the stubs are trusted, we are able to copy call arguments directly to the target domain. Traditional RPC implementations across address spaces typically perform three copies to transfer data. The arguments are marshalled into a message, the kernel copies the message to the target address space, and finally the callee must de-marshall the arguments. By having the caller and callee communicate via a shared buffer, LRPC also uses only a single copy to pass data between domains [BALL91].

The stubs are also responsible for managing machine state. On each cross-domain call any registers that are both used in the future by the caller and potentially modified by the callee must be protected. Only registers that are designated by architectural convention to be preserved across procedure calls are saved. As an optimization, if the callee domain contains no instructions that modify a preserved register we can avoid saving it. Karger uses a trusted linker to perform this kind of optimization between address spaces [Kar89]. In addition to saving and restoring registers, the stubs must switch the execution stack, establish the correct register context for the software encapsulation technique being used, and validate all dedicated registers.

Our system must also be robust in the presence of fatal errors, for example, an addressing violation, while executing in a fault domain. Our current implementation uses the UNIX signal facility to catch these errors; it then terminates the outstanding call and notifies the caller's fault domain. If the application uses the same operating system thread for all fault domains, there must be a way to terminate a call that is taking too long, for example, because of an infinite loop. Trusted modules may use a timer facility to interrupt execution periodically and determine if a call needs to be terminated.

5 Performance Results

To evaluate the performance of software-enforced fault domains, we implemented and measured a prototype of our system on a 40MHz DECstation 5000/240 (DEC-MIPS) and a 160Mhz Alpha 400 (DEC-ALPHA).

We consider three questions. First, how much overhead does software encapsulation incur? Second, how fast is a cross-fault-domain RPC? Third, what is the performance impact of using software enforced fault isolation on an end-user application? We discuss each of these questions in turn.

5.1 Encapsulation Overhead

We measured the execution time overhead of sandboxing a wide range of C programs, including the C SPEC92 benchmarks and several of the Splash benchmarks [Ass91, SWG91]. We treated each benchmark as if it were a distrusted module, sandboxing all of its code. Column 1 of Table 1 reports overhead on the DEC-MIPS, column 6 reports overhead on the DEC-ALPHA. Columns 2 and 7 report the overhead of using our technique to provide general protection by sandboxing load instructions as well as store and jump instructions[4]. As detailed in Section 3, sandboxing requires 5 dedicated registers. Column 3 reports the overhead of removing these registers from possible use by the compiler. All overheads are computed as the additional execution time divided by the original program's execution time.

On the DEC-MIPS, we used the program measurement tools `pixie` and `qpt` to calculate the number of additional instructions executed due to sandboxing [Dig, BL92]. Column 4 of Table 1 reports this data as a percentage of original program instruction counts.

The data in Table 1 appears to contain a number of anomalies. For some of the benchmark programs, for example, `056.ear` on the DEC-MIPS and `026.compress` on the DEC-ALPHA, sandboxing *reduced* execution time. In a number of cases the overhead is surprisingly low.

To identify the source of these variations we developed an analytical model for execution overhead. The model predicts overhead based on the number of additional instructions executed due to sandboxing (*s-instructions*), and the number of saved floating point interlock cycles (*interlocks*). Sandboxing increases the available instruction-level parallelism, allowing the number of floating-point interlocks to be substantially reduced. The integer pipeline does not provide interlocking; instead, delay slots are explicitly filled with nop instructions by the compiler or assembler. Hence, scheduling effects among integer instructions will be accurately reflected by the count of instructions added (*s-instructions*). The expected overhead is computed as:

$$\frac{(\textit{s-instructions} - \textit{interlocks}) / \textit{cycles-per-second}}{\textit{original-execution-time-seconds}}$$

[4]Loads in the libraries, such as the standard C library, were not sandboxed.

The model provides an effective way to separate known sources of overhead from second order effects. Column 5 of Table 1 are the predicted overheads.

As can be seen from Table 1, the model is, on average, effective at predicting sandboxing overhead. The differences between measured and expected overheads are normally distributed with mean 0.7% and standard deviation of 2.6%. The difference between the means of the measured and expected overheads is not statistically significant. This experiment demonstrates that, by combining instruction count overhead and floating point interlock measurements, we can accurately predict average execution time overhead. If we assume that the model is also accurate at predicting the overhead of individual benchmarks, we can conclude that there is a second order effect creating the observed anomalies in measured overhead.

We can discount effective instruction cache size and virtual memory paging as sources for the observed execution time variance. Because sandboxing adds instructions, the effective size of the instruction cache is reduced. While this might account for measured overheads higher than predicted, it does not account for the opposite effect. Because all of our benchmarks are compute bound, it is unlikely that the variations are due to virtual memory paging.

The DEC-MIPS has a physically indexed, physically tagged, direct mapped data cache. In our experiments sandboxing did not affect the size, contents, or starting virtual address of the data segment. For both original and sandboxed versions of the benchmark programs, successive runs showed insignificant variation. Though difficult to quantify, we do not believe that data cache alignment was an important source of variation in our experiments.

We conjecture that the observed variations are caused by *instruction cache mapping conflicts*. Software encapsulation changes the mapping of instructions to cache lines, hence changing the number of instruction cache conflicts. A number of researchers have investigated minimizing instruction cache conflicts to reduce execution time [McF89, PH90, Sam88]. One researcher reported a 20% performance gain by simply changing the order in which the object files were linked [PH90]. Samples and Hilfinger report significantly improved instruction cache miss rates by rearranging only 3% to 8% of an application's basic blocks [Sam88].

Beyond this effect, there were statistically significant differences among programs. On average, programs which contained a significant percentage of floating point operations incurred less overhead. On the DEC-MIPS the mean overhead for floating point intensive benchmarks is 2.5%, compared to a mean of 5.6% for the remaining benchmarks. All of our benchmarks are

Benchmark		DEC-MIPS					DEC-ALPHA	
		Fault Isolation Overhead	Protection Overhead	Reserved Register Overhead	Instruction Count Overhead	Fault Isolation Overhead (predicted)	Fault Isolation Overhead	Protection Overhead
052.alvinn	FP	1.4%	33.4%	-0.3%	19.4%	0.2%	8.1%	35.5%
bps	FP	5.6%	15.5%	-0.1%	8.9%	5.7%	4.7%	20.3%
cholesky	FP	0.0%	22.7%	0.5%	6.5%	-1.5%	0.0%	9.3%
026.compress	INT	3.3%	13.3%	0.0%	10.9%	4.4%	-4.3%	0.0%
056.ear	FP	-1.2%	19.1%	0.2%	12.4%	2.2%	3.7%	18.3%
023.eqntott	INT	2.9%	34.4%	1.0%	2.7%	2.2%	2.3%	17.4%
008.espresso	INT	12.4%	27.0%	-1.6%	11.8%	10.5%	13.3%	33.6%
001.gcc1.35	INT	3.1%	18.7%	-9.4%	17.0%	8.9%	NA	NA
022.li	INT	5.1%	23.4%	0.3%	14.9%	11.4%	5.4%	16.2%
locus	INT	8.7%	30.4%	4.3%	10.3%	8.6%	4.3%	8.7%
mp3d	FP	10.7%	10.7%	0.0%	13.3%	8.7%	0.0%	6.7%
psgrind	INT	10.4%	19.5%	1.3%	12.1%	9.9%	8.0%	36.0%
qcd	FP	0.5%	27.0%	2.0%	8.8%	1.2%	-0.8%	12.1%
072.sc	INT	5.6%	11.2%	7.0%	8.0%	3.8%	NA	NA
tracker	INT	-0.8%	10.5%	0.4%	3.9%	2.1%	10.9%	19.9%
water	FP	0.7%	7.4%	0.3%	6.7%	1.5%	4.3%	12.3%
Average		**4.3%**	**21.8%**	**0.4%**	**10.5%**	**5.0%**	**4.3%**	**17.6%**

Table 1: Sandboxing overheads for DEC-MIPS and DEC-ALPHA platforms. The benchmarks 001.gcc1.35 and 072.sc are dependent on a pointer size of 32 bits and do not compile on the DEC-ALPHA. The predicted fault isolation overhead for cholesky is negative due to conservative interlocking on the MIPS floating-point unit.

compute intensive. Programs that perform significant amounts of I/O will incur less overhead.

5.2 Fault Domain Crossing

We now turn to the cost of cross-fault-domain RPC. Our RPC mechanism spends most of its time saving and restoring registers. As detailed in Section 4, only registers that are designated by the architecture to be *preserved* across procedure calls need to be saved. In addition, if no instructions in the callee fault domain modify a preserved register then it does not need to be saved. Table 2 reports the times for three versions of a NULL cross-fault-domain RPC. Column 1 lists the crossing times when all data registers are caller saved. Column 2 lists the crossing times when the preserved integer registers are saved. Finally, the times listed in Column 3 include saving all preserved floating point registers. In many cases crossing times could be further reduced by statically partitioning the registers between domains.

For comparison, we measured two other calling mechanisms. First, we measured the time to perform a C procedure call that takes no arguments and returns no value. Second, we sent a single byte between two address spaces using the pipe abstraction provided by the native operating system and measured the round-trip time. These times are reported in the last two columns of Table 2. On these platforms, the cost of cross-address-space calls is roughly three orders of magnitude more expensive than local procedure calls.

Operating systems with highly optimized RPC implementations have reduced the cost of cross-address-space RPC to within roughly two orders of magnitude of local procedure calls. On Mach 3.0, cross-address-space RPC on a 25Mhz DECstation 5000/200 is 314 times more expensive than a local procedure call [Ber93]. The Spring operating system, running on a 40Mhz SPARCstation2, delivers cross-address-space RPC that is 73 times more expensive than a local leaf procedure call [HK93]. Software enforced fault isolation is able to reduce the relative cost of cross-fault-domain RPC by an order of magnitude over these systems.

5.3 Using Fault Domains in POSTGRES

To capture the effect of our system on application performance, we added software enforced fault domains to the POSTGRES database management system, and measured POSTGRES running the Sequoia 2000 benchmark [SFGM93]. The Sequoia 2000 benchmark

Platform	Cross Fault-Domain RPC			C Procedure Call	Pipes
	Caller Save Registers	Save Integer Registers	Save Integer+Float Registers		
DEC-MIPS	1.11μs	1.81μs	2.83μs	0.10μs	204.72μs
DEC-ALPHA	0.75μs	1.35μs	1.80μs	0.06μs	227.88μs

Table 2: Cross-fault-domain crossing times.

Sequoia 2000 Query	Untrusted Function Manager Overhead	Software-Enforced Fault Isolation Overhead	Number Cross-Domain Calls	DEC-MIPS-PIPE Overhead (predicted)
Query 6	1.4%	1.7%	60989	18.6%
Query 7	5.0%	1.8%	121986	38.6%
Query 8	9.0%	2.7%	121978	31.2%
Query 10	9.6%	5.7%	1427024	31.9%

Table 3: Fault isolation overhead for POSTGRES running Sequoia 2000 benchmark.

contains queries typical of those used by earth scientists in studying the climate. To support these kinds of non-traditional queries, POSTGRES provides a user-extensible type system. Currently, user-defined types are written in conventional programming languages, such as C, and dynamically loaded into the database manager. This has long been recognized to be a serious safety problem[Sto88].

Four of the eleven queries in the Sequoia 2000 benchmark make use of user-defined polygon data types. We measured these four queries using both unprotected dynamic linking and software-enforced fault isolation. Since the POSTGRES code is trusted, we only sandboxed the dynamically loaded user code. For this experiment, our cross-fault-domain RPC mechanism saved the preserved integer registers (the variant corresponding to Column 2 in Table 2). In addition, we instrumented the code to count the number of cross-fault-domain RPCs so that we could estimate the performance of fault isolation based on separate address spaces.

Table 3 presents the results. Untrusted user-defined functions in POSTGRES use a separate calling mechanism from built-in functions. Column 1 lists the overhead of the untrusted function manager without software enforced fault domains. All reported overheads in Table 3 are relative to original POSTGRES using the untrusted function manager. Column 2 reports the measured overhead of software enforced fault domains. Using the number of cross-domain calls listed in Column 3 and the DEC-MIPS-PIPE time reported in Table 2, Column 4 lists the estimated overhead using conventional hardware address spaces.

5.4 Analysis

For the POSTGRES experiment software encapsulation provided substantial savings over using native operating system services and hardware address spaces. In general, the savings provided by our techniques over hardware-based mechanisms is a function of the percentage of time spent in distrusted code (t_d), the percentage of time spent crossing among fault domains (t_c), the overhead of encapsulation (h), and the ratio, r, of our fault domain crossing time to the crossing time of the competing hardware-based RPC mechanism.

$$savings = (1 - r)t_c - ht_d$$

Figure 5 graphically depicts these trade-offs. The X axis gives the percentage of time an application spends crossing among fault domains. The Y axis reports the relative cost of software enforced fault-domain crossing over hardware address spaces. Assuming that the execution time overhead of encapsulated code is 4.3%, the shaded region illustrates when software enforced fault isolation is the better performance alternative.

Software-enforced fault isolation becomes increasingly attractive as applications achieve higher degrees of fault isolation (see Figure 5). For example, if an application spends 30% of its time crossing fault domains, our RPC mechanism need only perform 10% better than its competitor. Applications that currently spend as little as 10% of their time crossing require only a 39% improvement in fault domain crossing time. As reported in Section 5.2, our crossing time for the DEC-MIPS is 1.10μs and for the DEC-ALPHA 0.75μs. Hence,

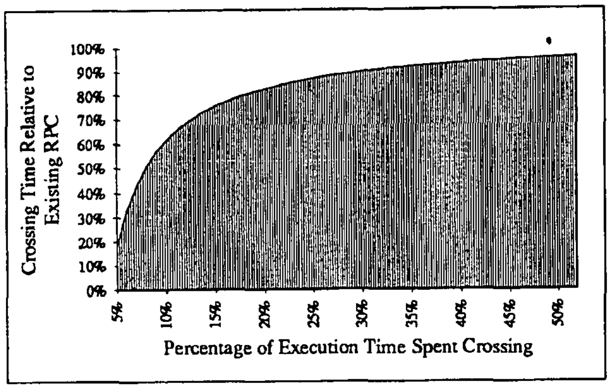

Figure 5: The shaded region represents when software enforced fault isolation provides the better performance alternative. The X axis represents percentage of time spent crossing among fault domains (t_c). The Y axis represents the relative RPC crossing speed (r). The curve represents the break even point: $(1-r)t_c = ht_d$. In this graph, $h = 0.043$ (encapsulation overhead on the DEC-MIPS and DEC-ALPHA).

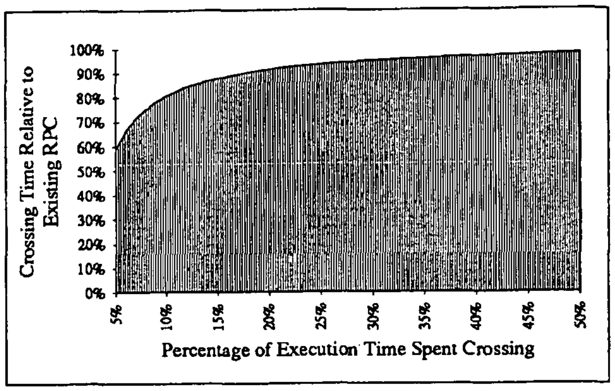

Figure 6: The shaded region represents when software enforced fault isolation provides the better performance alternative. The X axis represents percentage of time spent crossing among fault domains (t_c). The Y axis represents the relative RPC crossing speed (r). The curve represents the break even point: $(1-r)t_c = ht_d$. In this graph, $h = 0.043$ (encapsulation overhead on the DEC-MIPS and DEC-ALPHA).

for this latter example, a hardware address space crossing time of 1.80μs on the DEC-MIPS and 1.23μs on the DEC-ALPHA would provide better performance than software fault domains. As far as we know, no production or experimental system currently provides this level of performance.

Further, Figure 5 assumes that the entire application was encapsulated. For many applications, such as POSTGRES, this assumption is conservative. Figure 6 transforms the previous figure, assuming that 50% of total execution is spent in distrusted extension code.

Figures 5 and 6 illustrate that software enforced fault isolation is the best choice whenever crossing overhead is a significant proportion of an application's execution time. Figure 7 demonstrates that overhead due to software enforced fault isolation remains small regardless of application behavior. Figure 7 plots overhead as a function of crossing behavior and crossing cost. Crossing times typical of vendor-supplied and highly optimized hardware-based RPC mechanisms are shown. The graph illustrates the relative performance stability of the software solution. This stability allows system developers to ignore the performance effect of fault isolation in choosing which modules to place in separate fault domains.

6 Related Work

Many systems have considered ways of optimizing RPC performance [vvST88, TA88, Bla90, SB90, HK93, BALL90, BALL91]. Traditional RPC systems based

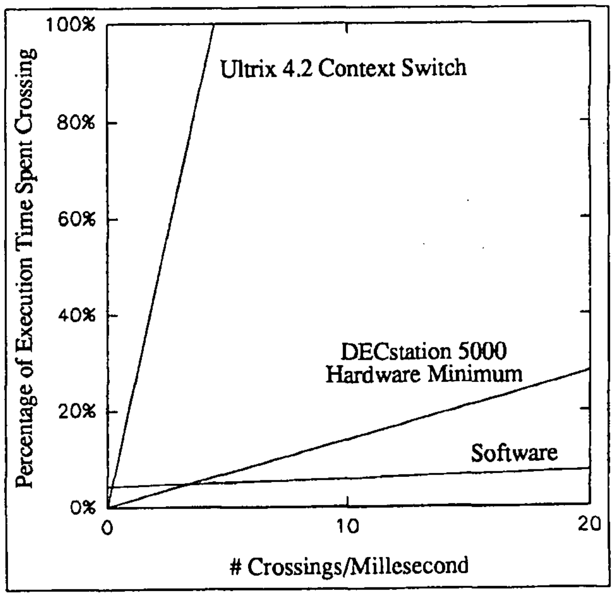

Figure 7: Percentage of time spent in crossing code versus number of fault domain crossings per millisecond on the DEC-MIPS. The hardware minimum crossing number is taken from a cross-architectural study of context switch times [ALBL91]. The Ultrix 4.2 context switch time is as reported in the last column of Table 2.

on hardware fault isolation are ultimately limited by the minimal hardware cost of taking two kernel traps and two hardware context switches. LRPC was one of the first RPC systems to approach this limit, and our prototype uses a number of the techniques found in LRPC and later systems: the same thread runs in both the caller and the callee domain, the stubs are kept as simple as possible, and the crossing code jumps directly to the called procedure, avoiding a dispatch in the callee domain. Unlike these systems, software-based fault isolation avoids hardware context switches, substantially reducing crossing costs.

Address space identifier tags can be used to reduce hardware context switch times. Tags allow more than one address space to share the TLB; otherwise the TLB must be flushed on each context switch. It was estimated that 25% of the cost of an LRPC on the Firefly (which does not have tags) was due to TLB misses[BALL90]. Address space tags do not, however, reduce the cost of register management or system calls, operations which are not scaling with integer performance[ALBL91]. An important advantage of software-based fault isolation is that it does not rely on specialized architectural features such as address space tags.

Restrictive programming languages can also be used to provide fault isolation. Pilot requires all kernel, user, and library code to be written in Mesa, a strongly typed language; all code then shares a single address space [RDH+80]. The main disadvantage of relying on strong typing is that it severely restricts the choice of programming languages, ruling out conventional languages like C, C++, and assembly. Even with strongly-typed languages such as Ada and Modula-3, programmers often find they need to use loopholes in the type system, undercutting fault isolation. In contrast, our techniques are language independent.

Deutsch and Grant built a system that allowed user-defined measurement modules to be dynamically loaded into the operating system and executed directly on the processor [DG71]. The module format was a stylized native object code designed to make it easier to statically verify that the code did not violate protection boundaries.

An interpreter can also provide failure isolation. For example, the BSD UNIX network packet filter utility defines a language which is interpreted by the operating system network driver. The interpreter insulates the operating system from possible faults in the customization code. Our approach allows code written in any programming language to be safely encapsulated (or rejected if it is not safe), and then executed at near full speed by the operating system.

Anonymous RPC exploits 64-bit address spaces to provide low latency RPC and *probabilistic* fault isolation [YBA93]. Logically independent domains are placed at random locations in the same hardware address space. Calls between domains are anonymous, that is, they do not reveal the location of the caller or the callee to either side. This provides probabilistic protection – it is unlikely that any domain will be able to discover the location of any other domain by malicious or accidental memory probes. To preserve anonymity, a cross domain call must trap to protected code in the kernel; however, no hardware context switch is needed.

7 Summary

We have described a software-based mechanism for portable, programming language independent fault isolation among cooperating software modules. By providing fault isolation within a single address space, this approach delivers cross-fault-domain communication that is more than an order of magnitude faster than any RPC mechanism to date.

To prevent distrusted modules from escaping their own fault domain, we use a software encapsulation technique, called *sandboxing*, that incurs about 4% execution time overhead. Despite this overhead in executing distrusted code, software-based fault isolation will often yield the best overall application performance. Extensive kernel optimizations can reduce the overhead of hardware-based RPC to within a factor of ten over our software-based alternative. Even in this situation, software-based fault isolation will be the better performance choice whenever the overhead of using hardware-based RPC is greater than 5%.

8 Acknowledgements

We thank Brian Bershad, Mike Burrows, John Hennessy, Peter Kessler, Butler Lampson, Ed Lazowska, Dave Patterson, John Ousterhout, Oliver Sharp, Richard Sites, Alan Smith and Mike Stonebraker for their helpful comments on the paper. Jim Larus provided us with the profiling tool qpt. We also thank Mike Olson and Paul Aoki for helping us with POSTGRES.

References

[ACD74] T.L. Adam, K.M. Chandy, and J.R. Dickson. A comparison of list schedules for parallel processing systems. *Communications of the ACM*, 17(12):685–690, December 1974.

[ALBL91] Thomas Anderson, Henry Levy, Brian Bershad, and Edward Lazowska. The Interaction of Architecture and Operating System Design.

In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 108–120, April 1991.

[Ass91] Administrator: National Computer Graphics Association. *SPEC Newsletter*, 3(4), December 1991.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, 1986.

[BALL90] Brian Bershad, Thomas Anderson, Edward Lazowska, and Henry Levy. Lightweight Remote Procedure Call. *ACM Transactions on Computer Systems*, 8(1), February 1990.

[BALL91] Brian Bershad, Thomas Anderson, Edward Lazowska, and Henry Levy. User-Level Interprocess Communication for Shared-Memory Multiprocessors. *ACM Transactions on Computer Systems*, 9(2), May 1991.

[Ber93] Brian Bershad, August 1993. Private Communication.

[BL92] Thomas Ball and James R. Larus. Optimally profiling and tracing. In *Proceedings of the Conference on Principles of Programming Languages*, pages 59–70, 1992.

[Bla90] David Black. Scheduling Support for Concurrency and Parallelism in the Mach Operating System. *IEEE Computer*, 23(5):35–43, May 1990.

[BN84] Andrew Birrell and Bruce Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[Cla92] J.D. Clark. *Window Programmer' Guide To OLE/DDE*. Prentice-Hall, 1992.

[DG71] L. P. Deutsch and C. A. Grant. A flexible measurement tool for software systems. In *IFIP Congress*, 1971.

[Dig] Digital Equipment Corporation. *Ultrix v4.2 Pixie Manual Page*.

[Dys92] Peter Dyson. Xtensions for Xpress: Modular Software for Custom Systems. *Seybold Report on Desktop Publishing*, 6(10):1–21, June 1992.

[FP93] Kevin Fall and Joseph Pasquale. Exploiting in-kernel data paths to improve I/O throughput and CPU a vailability. In *Proceedings of the 1993 Winter USENIX Conference*, pages 327–333, January 1993.

[HC92] Keiran Harty and David Cheriton. Application-controlled physical memory using external page-cache management. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1992.

[HK93] Graham Hamilton and Panos Kougiouris. The Spring nucleus: A microkernel for objects. In *Proceedings of the Summer USENIX Conference*, pages 147–159, June 1993.

[HKM$^+$88] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems*, 6(1):51–82, February 1988.

[Int86] Intel Corporation, Santa Clara, California. *Intel 80386 Programmer's Reference Manual*, 1986.

[JRT85] Michael B. Jones, Richard F. Rashid, and Mary R. Thompson. Matchmaker: An interface specification language for distributed processing. In *Proceedings of the 12th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 225–235, January 1985.

[Kar89] Paul A. Karger. Using Registers to Optimize Cross-Domain Call Performance. In *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 194–204, April 3-6 1989.

[Kle86] Steven R. Kleiman. Vnodes: An Architecture for Multiple File System Types in SUN UNIX. In *Proceedings of the 1986 Summer USENIX Conference*, pages 238–247, 1986.

[LB92] James R. Larus and Thomas Ball. Rewriting executable files to measure program behavior. Technical Report 1083, University of Wisconsin-Madison, March 1992.

[McF89] Scott McFarling. Program optimization for instruction caches. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 183–191, April 1989.

[MJ93] Steven McCanne and Van Jacobsen. The BSD Packet Filter: A New Architecture for User-Level Packet Capture. In *Proceedings of the 1993 Winter USENIX Conference*, January 1993.

[MRA87] J. C. Mogul, R. F. Rashid, and M. J. Accetta. The packet filter: An efficient mechanism for user-level network code. In *Proceedings of the Symposium on Operating System Principles*, pages 39–51, November 1987.

[PH90] Karl Pettis and Robert C. Hansen. Profile guided code positioning. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 16–27, White Plains, New York, June 1990. Appeared as SIGPLAN NOTICES 25(6).

[RDH$^+$80] David D. Redell, Yogen K. Dalal, Thomas R. Horsley, Hugh C. Lauer, William C. Lynch, Paul R. McJones, Hal G. Murray, and Stephen C. Purcell. Pilot: An Operating System for a Personal Computer. *Communications of the ACM*, 23(2):81–92, February 1980.

[Sam88] A. Dain Samples. Code reorganization for instruction caches. Technical Report UCB/CSD 88/447, University of California, Berkeley, October 1988.

[SB90] Michael Schroeder and Michael Burrows. Performance of Firefly RPC. *ACM Transactions on Computer Systems*, 8(1):1–17, February 1990.

[SCK+93] Richard L. Sites, Anton Chernoff, Matthew B. Kirk, Maurice P. Marks, and Scott G. Robinson. Binary translation. *Communications of the ACM*, 36(2):69–81, February 1993.

[SFGM93] M. Stonebraker, J. Frew, K. Gardels, and J. Meridith. The Sequoia 2000 Benchmark. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, May 1993.

[Sto87] Michael Stonebraker. Extensibility in POSTGRES. *IEEE Database Engineering*, September 1987.

[Sto88] Michael Stonebraker. Inclusion of new types in relational data base systems. In Michael Stonebraker, editor, *Readings in Database Systems*, pages 480–487. Morgan Kaufmann Publishers, Inc., 1988.

[SWG91] J. P. Singh, W. Weber, and A. Gupta. Splash: Stanford parallel applications for shared-memory. Technical Report CSL-TR-91-469, Stanford, 1991.

[TA88] Shin-Yuan Tzou and David P. Anderson. A Performance Evaluation of the DASH Message-Passing System. Technical Report UCB/CSD 88/452, Computer Science Division, University of California, Berkeley, October 1988.

[Thi92] Thinking Machines Corporation. *CM-5 Network Interface Programmer's Guide*, 1992.

[vCGS92] T. von Eicken, D. Culler, S. Goldstein, and K. Schauser. Active Messages: A Mechanism for Integrated Communication and Computation. In *Proceedings of the 19th Annual Symposium on Computer Architecture*, 1992.

[vvST88] Robbert van Renesse, Hans van Staveren, and Andrew S. Tanenbaum. Performance of the World's Fastest Distributed Operating System. *Operating Systems Review*, 22(4):25–34, October 1988.

[Web93] Neil Webber. Operating System Support for Portable Filesystem Extensions. In *Proceedings of the 1993 Winter USENIX Conference*, January 1993.

[YBA93] Curtis Yarvin, Richard Bukowski, and Thomas Anderson. Anonymous RPC: Low Latency Protection in a 64-Bit Address Space. In *Proceedings of the Summer USENIX Conference*, June 1993.